Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche
researchportal.unamur.be
University of Namur

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Contribution à l'étude de modèles de cycle de vie en programmation impérative et logique

De Raedemacker, Philippe

Award date:
1989

Awarding institution:
Universite de Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 20. Apr. 2024

FACULTES
UNIVERSITAIRES
N.D. DE LA PAIX
NAMUR

INSTITUT D'INFORMATIQUE

CONTRIBUTION A L'ETUDE DE MODELES DE CYCLE DE VIE EN PROGRAMMATION IMPERATIVE ET LOGIQUE.

Philippe De Raedemacker
Promoteur : Professeur Baudouin Le Charlier

Mémoire présenté en vue de l'obtention du titre
de Licencié et Maîtrise en Informatique
Année Académique 1988-1989.

RUE GRANDGAGNAGE, 21, B - 5000 NAMUR (BELGIUM)

RESUME.

Nous décrivons, dans une première partie de ce travail, les différentes étapes qui ont conduit à la construction d'un outil d'aide à la programmation impérative. Nous présentons essentiellement l'identification des objets logiciels et la construction d'un schéma Entité/Relation mettant en évidence les caractéristiques de ces objets ainsi que leurs relations. Enfin, nous précisons les fonctionnalités principales de notre outil. Dans une seconde partie, nous appliquons ce même cadre de travail à la programmation logique. Nous pouvons ainsi décrire, par un modèle Entité/Relation, la méthodologie de développement de programme PROLOG. Nous spécifions alors un système de contrôle des objets évoluant en versions multiples.

ABSTRACT.

We describe, in a first part of this work, different steps that conclude to the building of an imperative programming support tool.

We emphasize especially the identification process of software objects and the construction of an Entity/Association diagram which highlights the objects's characteristic and their relationships. Finally, we make clear the main functionnalities of our tool. In a second part, we apply this framework to logic programming. Thus, we describe, by an Entity/Association diagram too, the methodology for systematic logic programm development. Then, we specify a control system for multiple versions of objects.

Je tiens à remercier :

Monsieur Baudoin Le Charlier, Professeur à l'Institut d'Informatique de l'Université de Namur, qui a bien voulu diriger ce mémoire.

Monsieur Yves Pigneur, Professeur à l'Université de Lausanne (Suisse) qui a accepté de coordonner mon stage de fin détude à l'Université de Lausanne.
Qu'il soit remercié pour les nombreux conseils qui m'ont permis de mener à bien ce mémoire.

Monsieur Yves Deville, chargé de recherche à l'Institut d'Informatique, dont les idées ont largement contribué à la réussite de ce travail.

Monsieur Pierre Flener, membre du projet de recherche FOLON à l'Institut d'Informatique, pour l'intérêt qu'il a manifesté pour ce travail. Qu'il trouve ici l'expression de ma reconnaissance pour avoir accepté de lire et corriger les versions de ce document.

Qu'il me soit permis d'exprimer ma profonde gratitude à tous les professeurs de l'Institut d'Informatique pour toutes les connaissances reçues et la formation apportée durant mon cycle d'études.

CHAPITRE 1 :

TENDANCES ACTUELLES DES MODELES DE CYCLE DE VIE.

1.1. Introduction.

Avec le développement des techniques apparu dès le début des années 60, l'informatique a vu son champ d'application s'étendre à des problèmes de plus en plus complexes comme par exemple l'informatisation des procédures de lancement des navettes spatiales, le contrôle de processus industriels...
Les logiciels conçus à cet effet sont la plupart volumineux (on parle de 80.000.000 instructions objet pour la station de l'espace) et ont des spécifications complexes. Il est alors rapidement apparu que la demande de logiciels augmentait plus rapidement que notre capacité à les produire et à les maintenir.

Dès la fin des années 60, une série de conférences furent organisées pour débattre de problèmes liés au développement de logiciels, mais qui n'ont pas varié depuis lors, à savoir : le coût de développement, le manque de fiabilité et la difficulté de maintenance d'un logiciel. Le terme de génie logiciel était ainsi créé.

La terminologie du "IEEE Standard Glossary of Software Engineering" définit le génie logiciel comme ..."une approche systématique pour le développement, l'exploitation, la maintenance et l'abandon d'un logiciel..." (IEEE 83).

Les objectifs principaux du génie logiciel sont donc l'amélioration de la qualité des logiciels produits et l'augmentation de la productivité.

Avant d'introduire un certain nombre de techniques qui permettent d'atteindre les objectifs du génie lociciel, il nous a semblé intéressant de clarifier ces buts.

La qualité d'un logiciel peut être regardée selon plusieurs approches, décrites par une série de facteurs. D'une part, nous pouvons dégager des facteurs de qualité externes, c'est-à-dire des facteurs détectés par l'utilisateur final ou l'utilisateur chargé de l'achat du logiciel. Typiquement nous retrouverons principalement dans cette catégorie la correction d'un logiciel, sa robustesse, son extensibilité, sa réutilisabilité, sa comptabilité, mais aussi son efficacité, sa portabilité, son intégrité, sa vérifiabilité et sa facilité d'utilisation. Nous pouvons, d'autre part, définir des qualités internes, c'est-à-dire des facteurs perçus par les analystes/programmeurs du logiciel. Cette catégorie reprend la modularité et la lisibilité.

La productivité peut, quant à elle, être mesurée par le rapport entre les sorties produites et les entrées consommées. Par "entrées", nous entendons le travail des développeurs et tout le matériel utilisé pour le développement d'un logiciel. Les "sorties" peuvent être calculées par le nombre de lignes de code source du logiciel produit.

Face à ces problèmes, le génie logiciel s'est attaché à mettre au point et à appliquer une méthodologie de développement de logiciel qui doit, d'une part, faire l'inventaire de toutes les tâches à entreprendre pour produire un logiciel de qualité et d'autre part, couvrir tous les aspects de gestion des ressources à allouer au projet logiciel. Le principal facteur permettant de rendre les différentes étapes du développement de logiciel plus efficaces réside dans l'application d'outils logiciels assistant et/ou automatisant les parties répétitives et fastidieuses de chaque étape du cycle de vie d'un logiciel.

Si, en 1986, le montant mondial des coûts des logiciels était estimé à 140 milliards de dollars, les perpectives (Boehm 87), sur base d'une augmentation de 12% l'an, annoncent un total de 450 milliards de dollars en 1995.
Une amélioration de la productivité de l'ordre de 20% pourrait faire gagner 90 milliards de dollars. Ce gain substantiel vaut à lui seul les efforts entrepris pour réaliser ces objectifs !

Le plan de ce travail se décompose comme suit :

- La suite du chapitre 1 fera le point sur des travaux menés dans le cadre des modèles de cycle de vie de logiciel. On tentera ensuite d'éclairer quelques caractéristiques propres à tout atelier logiciel moderne.
- Le chapitre 2 est basé sur le travail effectué dans le cadre de notre stage de fin d'étude à l'Université de Lausanne auprès du Professeur Yves Pigneur. Il présente un modèle de cycle de vie simplifié pour la programmation impérative.

  Une modélisation selon les concepts du modèle Entité/ Relation de ce cycle de vie et les spécifications d'un outil d'aide à la programmation impératiive sont ensuite exposées. Enfin, une présentation de la conversion du

schéma Entité/Relation en une base de données relationnelle et une brève présentation de l'architecture de l'outil précédent une conclusion de ce chapitre.

- Le chapitre 3 se fonde sur la thèse de doctorat de Yves Deville, chargé de recherche à l'Institut d'Informatique, concernant une méthodologie de développement de logiciel en programmation logique. Ce chapitre expose une modélisation de cette méthode de construction selon les concepts du modèle Entité/Relation. Une dicussion des fonctionnalités d'un outil d'aide à la gestion des informations mises en lumière dans le schéma Entité/Relation est ensuite présentée. Enfin, ce chapitre se conclut par une évolution du travail réalisé. Cette recherche a été réalisée dans le cadre du projet FOLON.

1.2. DES MODELES DE CYCLE DE VIE.

La fonction principale du cycle de vie d'un logiciel est double.
D'une part, il doit définir et ordonner les différentes phases de développement et d'évolution d'un logiciel. D'autre part, il doit dégager les critères de transition d'une phase à la suivante. Un modèle de cycle de vie doit donc répondre aux deux questions suivantes :

- Que devons nous faire ensuite ?
- Combien de temps devons nous le faire ?

Cette section se propose d'illustrer les réalisations passées ainsi que les tendances actuelles dans cet important domaine.

1.2.1. Le modèle "Code and Fix".

Au début du développement de projets logiciels, le modèle suivi se résumait à deux étapes :

- écrire du code,
- corriger les problèmes du code.

Les difficultés essentielles rencontrées avec ce genre de modèle sont :

- Après plusieurs corrections, le code devient si peu structuré que les modifications ultérieures sont très coûteuses.
- Assez souvent, même un logiciel bien conçu ne correspond pas aux besoins de l'utilisateur, le logiciel est alors soit rejeté, soit redéveloppé.
- Le code ne se prête ni aux tests, ni aux modifications.

Ces quelques problèmes mettent en lumière l'utilité d'une phase d'analyse des besoins et d'une phase de conception avant le codage. Ils soulignent en outre la nécessité de préparer des tâches de tests dans les phases préliminaires.

1.2.2. Le modèle en cascade (Waterfall Model).

Le modèle en cascade, encore appelé dans la littérature le modèle conventionnel de cycle de vie, est composé de différentes étapes se succédant dans une séquence souvent immuable, avec une validation de chaque étape et une possibilité de feedback entre deux étapes successives.

Ce modèle est présenté à la figure 1.1 suivante :

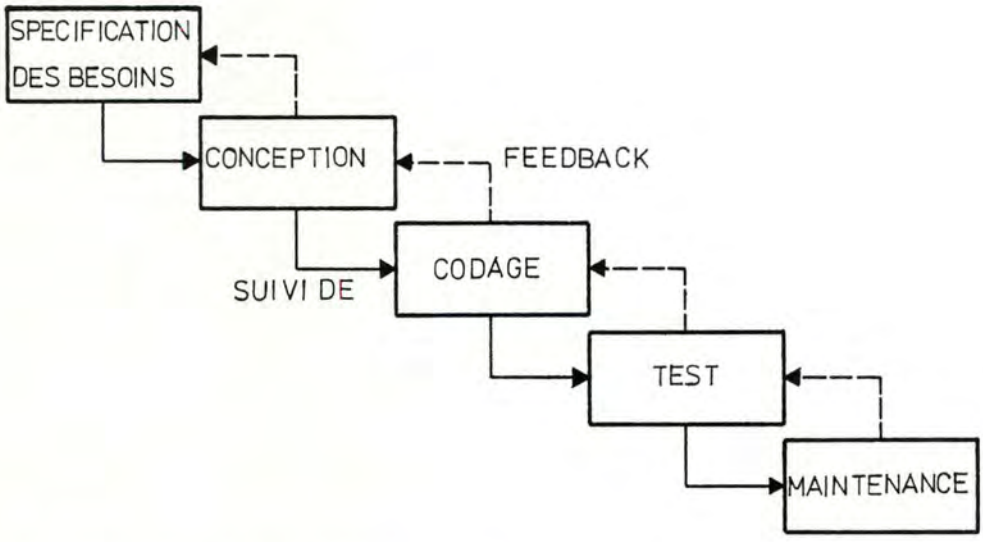


FIGURE 1.1 MODELE CONVENTIONNEL "EN CASCADE" DU CYCLE DE VIE D'UN LOGICIEL.

Un premier modèle a été proposé par Royce (Royce 70); depuis il subit de nombreuses modifications (Boehm 76 & 81). Mais nous pouvons mettre clairement en évidence ses principales activités et leurs résultats :

- Spécification des besoins - Elle consiste en un énoncé des fonctions que le logiciel doit réaliser, c'est-à-dire une description précise, complète et cohérente du comportement du logiciel,
- Conception - Elle décrit comment le logiciel satisfait aux besoins, c'est-à-dire la structure interne des éléments qui exécutent les fonctions,

- Codage - Elle correspond à l'implantation du résultat de la conception dans un langage de programmation,
- Test - Cette activité se résume à la recherche et à la correction d'un maximum d'erreurs du logiciel, et à la validation de ce logiciel avant sa mise en service,
- Maintenance - Cette dernière activité correspond à la recherche et la correction des erreurs et à l'adaptation du logiciel aux nouveaux matériels et aux évolutions du système lui-même.

Il convient de mettre en évidence quelques inconvénients du modèle conventionnel. Un premier problème à souligner est le besoin d'une grande quantité de documents très élaborés lors des phases de spécialisation et de conception.
Cette approche "dirigée par les documents" n'est pas toujours pertinente pour des applications interactives.

Un second problème réside dans l'intervention tardive de l'utilisateur final dans le cycle de vie ou plus tôt après la phase de codage . Si le système n'est pas conforme à l'attente de l'utilisateur, un réexamen des spécifications, de la conception et du codage peut être coûteuse.

Les points suivants présentent des alternatives pouvant surmonter ces quelques critiques.

1.2.3. Le prototypage.

Le prototypage peut être défini comme un processus de construction d'un modèle de travail d'un logiciel ou d'une partie d'un logiciel (Taylor 82), (Scharer 83). Son objectif est de clarifier les caractéristiques et les opérations d'un logiciel en construisant une version de démonstration ayant moins de fonctionnalités ou de moins bonnes performances que le logiciel lui-même.

Cet objectif mis-à-jour, l'utilité d'introduire le concept de prototypage dans le développement d'un logiciel prend tout son sens. Ainsi, le prototypage aide le développeur du logiciel lorsque les exigences de l'utilisateur ne sont pas très pécises. Cette situation, très courante dans la pratique est peu supportée par le modèle conventionnel. Le prototypage facilite donc les rapports entre les développeurs et les utilisateurs en se concentrant sur le comportement réel du logiciel et les expériences empiriques de l'utilisateur.

On peut donc définir un modèle de développement tenant compte de ce nouveau concept.
Un tel modèle est décrit par la Figure 1.2 suivante :

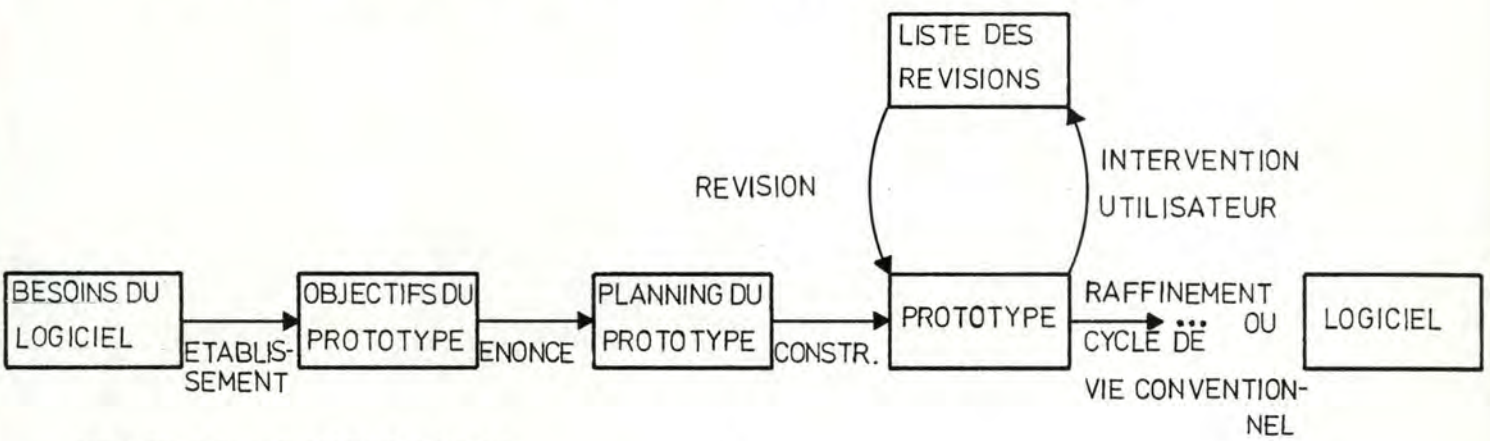


FIGURE 1.2 LE PROTOTYPAGE.

En partant de l'expression des besoins souvent incomplets et/ou vagues de l'utilisateur, l'analyste détermine l'objectif, l'énoncé du travail à réaliser par le prototype. Ceci lui permettra de mieux appréhender la faisabilité de la conception, les questions de performances, les besoins de l'utilisateur dans un domaine particulier.

Cet énoncé du travail du prototype sert de base au planning du développement de celui-ci. En effet, ce développement peut être coûteux et long, un planning efficace assure la livraison du prototype en temps voulu. Le projet de développement pourrait par exemple, maximiser l'utilisation de langages de très haut niveau ou d'utilitaires de gestion d'écran pour accélérer le processus.

Les itérations sur le prototype proviennent directement de la réponse de l'utilisateur. S'il n'y a pas d'itération, l'analyste a atteint l'objectif assigné au prototype. Une boucle signifie qu'une révision du prototype est nécessaire pour obtenir l'information supplémentaire.

Les objectifs du prototype une fois atteints, on peut décider soit de le raffiner en un logiciel complet, soit de commencer un processus de développement conventionnel bénéficiant du prototype déjà construit. Cette décision peut être prise en évaluant ces deux solutions selon les deux critères suivants :

- Combien de fonctionnalités sont déjà présentes dans le prototype ?
- Le prototype vaut-il la peine que l'on y investisse encore des efforts ?

1.2.4. Les spécifications opérationnelles.

Une spécification opérationnelle (Zave 84) peut être définie comme un modèle du logiciel qui peut être évalué ou exécuté afin de reproduire le comportement du logiciel.
Cette spécification est développée lors de la phase d'analyse des besoins; mais de par sa nature exécutable, elle diffère d'une spécification des besoins habituelle.

Rappelons qu'une spécification "normale" s'intéresse aux fonctionnalités du système et décrit "ce que" le logiciel devra faire et pas "comment" il le fait. La conception quant à elle exprime la structure interne qui doit permettre de répondre aux fonctionnalités. Le cycle de vie conventionnel tient compte de la séparation de ces deux activités. Il est pourtant parfois difficile de les différencier. Une conception doit, en effet, tenir compte aussi bien de la décomposition en fonctions que par exemple des contraintes de performance de l'environnement hardware/software cible.

De par sa nature exécutable, la construction opérationnelle invite à une nouvelle définition de la séparation de ces deux activités. Cette approche essaye de séparer les questions liées au problème et celles liées à l'implémentation. Son but est de produire une spécification qui ne traite que des questions inhérentes au problème traité et de pouvoir exprimer cette spécification sous une forme exécutable permettant de juger le comportement du logiciel. Dans ce sens, une spécification opérationnelle est comparable à un prototypage. le comportement ainsi généré ne réflète pas la réalité

de l'environnement cible. Ce modèle est décrit à la Figure 1.3.

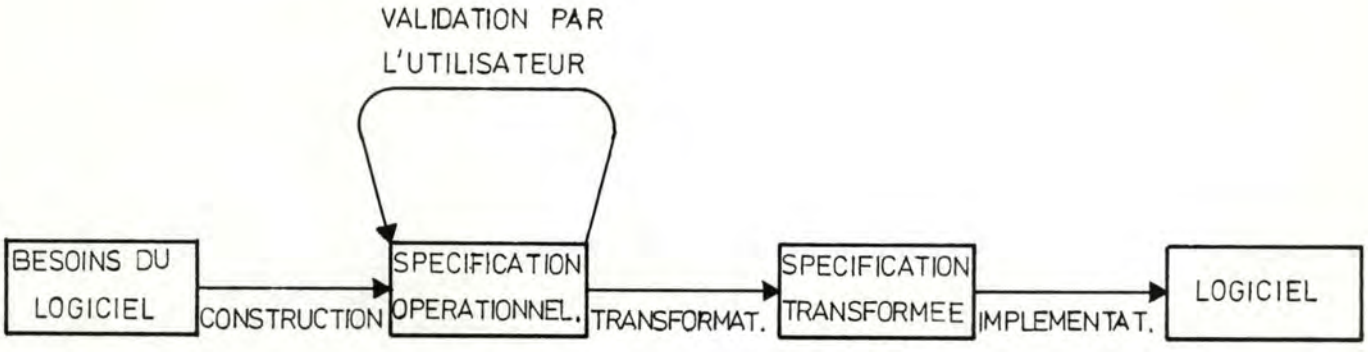


FIGURE 1.3 LES SPECIFICATIONS OPERATIONNELLES.

A partir des besoins du logiciel, l'analyste construit une spécification opérationnelle. Une fois celle-ci validée et/ou modifiée, l'analyste peut soit la considérer comme un prototype et revenir au cycle de vie en cascade en commençant la phase de conception (dans ce cas il perd la puissance du modèle), soit peut transformer les mécanismes qui généraient le comportement du logiciel pour les faire correspondre à l'environnement cible. Ce n'est qu'à ces niveaux que sont débattues les questions relatives à l'implémentation.

Une dernière phase fait correspondre les spécifications transformées à une implémentation de la solution dans le langage de programmation cible.

Nous résumerons les avantages de ce modèle en constatant

- la séparation entre les questions dirigées par le problème et celles de l'implémentation,
- la construction d'un modèle du logiciel clarifiant les besoins de l'utilisateur et pouvant être validé au début du développement.

Pour le lecteur intéressé par le sujet, nous nous bornerons à citer quelques langages permettant l'expansion de spécifications opérationnelles : GIST (Balzer 82), Paisley (Zave 82).

1.2.5. L'implémentation transformationnelle.

Cette approche (Partsch 83), (Balzer 81) utilise des supports automatisés pour appliquer une série de transformations qui changent une spécification en un système logiciel concret satisfaisant à cette spécification. Les différentes phases d'un modèle de cycle de vie intégrant ce concept sont représentées sur la Figure 1.4 suivante :

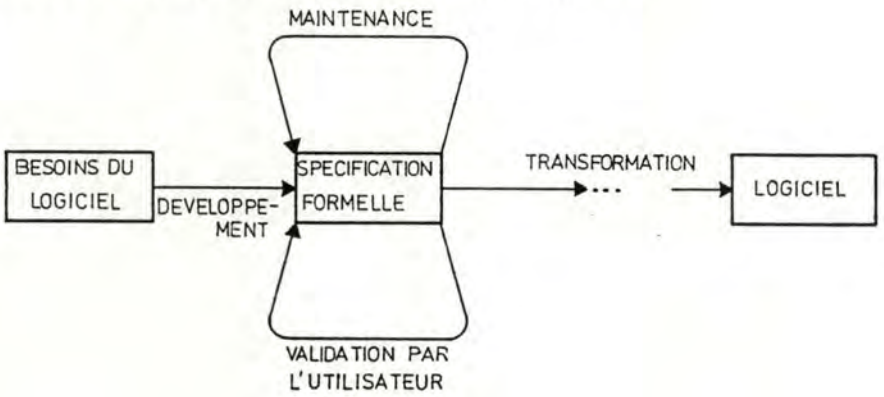


FIGURE 1.4 L'IMPLEMENTATION TRANSFORMATIONNELLE.

La première étape consiste à développer une spécification formelle à partir des besoins du logiciel. Cette étape est problématique car d'une part les spécifications doivent être définies d'une manière précise pour permettre l'application de transformations. D'autre part, elles doivent être compréhensibles pour les utilisateurs pour qu'ils puissent aisément les valider par rapport à leur attente.

Cette spécification formelle est la base de ce modèle car elle est l'objet d'une validation par rapport aux besoins de l'utilisateur. Elle est, ensuite, le point de départ d'une phase de transformation. Enfin elle est l'objet de la maintenance, car une modification des fonctionnalités s'exprime par la modification de la spécification et non du code.

Les transformations à appliquer peuvent correspondre, par exemple à la spécialisation ou à la généralisation de la représentation, au choix d'une structure de données, à la modification de la structure pour améliorer l'efficacité. Toutes les règles de transformation doivent préserver la cohérence des spécifications.

Quelques perpectives intéressantes à l'utilisation d'un tel modèle vont être signalées :

- l'automatisation de l'application des règles de transformation permet de réduire le travail de l'analyste
- l'application des règles de transformation exprimables formellement assure la cohérence des spécifications,

- l'élimination d'une phase particulière de test et son remplacement par une validation des spécifications par l'utilisateur.

1.2.6. Conclusions.

Les trois modèles de cycle de vie qui viennent d'être présenté, le prototypage, les spécificatons opérationnelles et l'implémentation transformationnelle, peuvent être présentés comme des alternatives au modèle de cycle de vie en cascade.

Nous soulignerons quelques relations évidentes entre ces modèles :

- la spécification formelle (Figure 1.4) peut être vue comme une spécification opérationnelle,
- l'affinage d'une spécification opérationnelle peut être automatisé,
- une spécification opérationnelle (Figure 1.3) est un prototype.

Ces trois modèles sont construits autour d'un même principe, fournir une version exécutable du logiciel à l'utilisateur le plus rapidement possible et rendre la production du logiciel plus automatique.

1.3. LES ATELIERS LOGICIELS.

Des travaux de recherche ont mis en évidence le besoin de formalismes et d'outils pour maîtriser les problèmes de la programmation. On escompte ainsi obtenir des accroissements substantiels en productivité et en qualité du logiciel développé. Le but d'un atelier logiciel est de fournir un ensemble intégré et cohérent d'outils permettant d'automatiser les nombreux aspects non créatifs du développement et de la maintenance de logiciels par de multiples personnes et en de multiples versions.

Nous allons tenter de dégager un certain nombre de propriétés qui devraient caractériser un atelier logiciel se voulant productif et utilisable à grande échelle.
Certaines de ces propriétés sont discutées dans (Osterweil 81) (van Lamsweerde 82 & 88).

1.3.1. Intégration d'outils autonomes.

L'utilisateur d'un atelier doit avoir accès à un ensemble cohérent d'outils compatibles, complémentaires et ne réalisant qu'une seule fonction. Une telle intégration ne peut reposer que sur l'utilisation d'une structure de données commune à tous les outils, les fichiers par exemple. Une autre approche, qui devient actuellement indispensable à tout atelier, est d'articuler les outils autour d'une base de données centralisant toutes les informations relatives au projet logiciel.

Nous rappelons brièvement les caractéristiques de trois modèles de base de données :

1.3.1.1. le modèle relationnel.

Le modèle relationnel (Codd 70) structure les informations pertinentes concernant les différentes étapes du cycle de vie sous forme de relations.

Une relation est un ensemble de n - uplets où n est appelé degré de la relation. Chaque élément d'un cycle, appelé attribut, est nécessairement d'un type prédéfini : entier, réel, booléen, chaîne. Des contraintes peuvent aussi être exprimées sur les relations et les attributs, elles garantissent la cohérence des informations. Nous signalerons l'existence de langages de manpulation comme par exemple les langages algébriques utilisant l'union, l'intersection, la différence, la sélection, la projection et la jointure. Ils permettent d'exprimer des requêtes sur les relations sans se soucier de la manière de rechercher des informations.

1.3.1.2. le modèle Entité /Relation.

Le modèle Entité /Relation (Chen 76), (Bodart 83), (Gallo 86), (Penedo 85 & 87), (Van Lamsweerde 86) introduit une séparation entre les entités. Une entité caractérise un objet considéré comme pertinent pour le cycle de vie. Une relation représente un lien entre entités. Ce modèle a introduit une distinction entre les propriétés,

appelé parfois attributs, d'une entité et celles d'une relation entre entités.

1.3.1.3. le modèle orienté objet.

Les modèles orientés objet (Horowitz 86), (Meyer 88) considèrent la base comme un ensemble d'objets. Les attributs d'un objet peuvent être de types simples (entier, caractère,... etc) mais aussi des références vers un autre objet. Plutôt que de se concentrer sur un une série d'objets, nous pouvons regrouper les objets ayant des carctéristiques communes sous le concept de classe d'objets. Une classe décrit la structure statique d'un ensemble d'objets possibles. Tout objet d'une classe peut être appelé instance de la classe et correspond à un aspect dynamique. Ce modèle permet en outre de décrire un ensemble d'opérations qui peuvent être réalisées sur une instance d'une classe. Les relations entre objets sont exprimées au niveau des attributs de ces objets. Ainsi, une relation 1 - 1 correspond au mécanisme d'héritage (relation "is_a") et une relation 1 - N correspond au mécanisme client/serveur.

Beaucoup de produits élaborés tout au long du cycle de vie d'un logiciel peuvent être considérés comme des objets structurés selon les formalismes dans lesquels ils sont décrits. Ce fait nous permet d'induire un second principe d'intégration. Il

consiste à représenter tout objet formel manipulé dans l'atelier sous la forme unique d'arbre de syntaxe abstraite. Les outils manipulant de tels objets sont alors dirigés par la syntaxe des formalismes correspondants.

1.3.2. Couverture de la totalité du cycle de vie.

Un bon atelier devrait couvrir la totalité du cycle de vie d'un logiciel, aussi bien selon l'axe vertical (des spécifications fonctionnelles à la maintenance) que selon l'axe horizontal (ce dernier couvrant les activités et élaboration de produits, de validation, de documentation et de gestion de projet).

Malheureusement, fort peu d'ateliers existant à ce jour peuvent prétendre à ce souhait.

1.3.3. Interface agréable à l'utilisateur.

Cette propriété nécessite des capacités d'autodocumentation, des facilités de "HELP", un langage de commande simple, des représentations graphiques... etc. L'atelier devrait s'ajuster à la façon dont travaille habituellement l'utilisateur.

1.3.4. Atelier générique.

Il paraît essentiel, devant la multitude de langages et de méthodologies informatiques existants, qu'un atelier puisse être automatiquement adaptable aux langages et méthodologie propres à un utilisateur particulier. Alors qu'il y a de plus en plus d'environnements de programmation qui sont paramétrés sur les langages de programma-

tion, il n'existe qu'un nombre infime d'environnements de développement de logiciels qui sont paramétrés sur les modèles de cycle de vie sous-jacents.

La mise en oeuvre d'un atelier logiciel est donc de permettre de réduire les coûts de développement et de maintenance d'un logiciel et d'augmenter la qualité du logiciel produit. L'efficacité d'un atelier logiciel devrait être mesurée et évaluée.

CHAPITRE 2 :

MODELE DE CYCLE DE VIE EN PROGRAMMATION IMPERATIVE.

2.1. Introduction.

Force nous est de constater que, dans le processus de développement d'un logiciel, l'étape de codage est généralement reconnue comme étant une des activités les moins créatrices pour les membres d'un projet logiciel. Dans ce chapitre, nous allons décrire les diverses phases qui ont conduit à la construction d'un outil d'aide à la programmation en langue procédurale VAX/Pascal. Nous permettons ainsi aux développeurs de projet de programmer efficacement et de se concentrer sur des phases plus critiques, comme l'architecture logique par exemple.

Pour cadrer aux efforts de recherche active au sujet des outils d'aide au développement de logiciels, nous avons voulu construire un outil utilisant une base d'objets.
L'intérêt de structurer toutes les connaissances requises lors du développement d'un logiciel est présenté dans (Osterweil 81). Citons simplement qu'une telle base permet de maintenir les propriétés des composants du logiciel et de leurs relations.

Il est clair que, durant leur cycle de vie, les logiciels changent constamment et engendrent donc des composants de différentes natures, produits durant ce cycle de vie du logiciel (Boehm 76). Ces composants sont couramment désignés sous le terme générique d'objet logiciel. Ces objets

sont caractérisés par des attributs, se développent en versions multiples et ont des relations de différents types.

Une identification de ces composants n'est donc possible que par la définition précise du cycle de vie faisant évoluer le logiciel. La Section 2.2. va donc présenter un modèle de ce cycle de vie. Celui-ci couvre le passage d'une architecture logicielle à une forme exécutable.

La Section 2.3. va proposer une modélisation , par un schéma Entité/Relation (schéma E/R), du cycle de vie proposé. Elle nous permettra de mettre en évidence les caractéristiques des objets ainsi que leurs relations. Nous précisons, à la Section 2.4.,les fonctionnalités de notre outil.

Après avoir examiné la nature des informations pertinentes dans le cycle de vie d'un logiciel et les fonctions d'un outil d'aide à la programmation, nous allons présenter à la Section 2.5. le système développé pour supporter la base d'objets ainsi qu'une brève description de l'architecture de notre outil.

Les langages de programmation traditionnels, comme la plupart des Pascal, ne supportent pas directement la construction modulaire de programmes. Cependant, il nous semble intéressant de mettre en évidence le fait que le langage VAX/Pascal ne possède pas cette restriction.

Il offre, en effet, la possibilité de décomposer un programme en unités compilables séparément : les "programs" et les "modules". Un "program" peut être compilé, relié et exécuté seul. Un "module" ne peut, quant à lui, être exécuté sans être relié à un "program" mais peut être par contre compilé.

Pour permettre à des unités compilées séparément de communiquer, le langage de programmation VAX/Pascal dispose de deux mécanismes de partage des déclarations. Ceux-ci sont complètement décrits dans (DEC/PAS), seule une présentation des grands principes va être faite.

Le premier mécanisme permet aux unités compilables de se partager uniquement des variables, des procédures ou des fonctions par l'intermédiaire des attributs "[GLOBAL]" et "[EXTERNAL]". Cette technique est la seule qui permet à des unités écrites dans des langages différents de se partager les déclarations. Cependant, le compilateur n'assure pas la vérification de la concordance des types des variables.

Le principe général de ce mécanisme est maintenant décrit. Les variables et les routines (procédures ou fonctions), déclarées dans une unité de compilation, peuvent être référencées par une autre unité de compilation si les déclarations partagées de la prémière unité incluent l'attribut "[GLOBAL]" et si les déclarations partagées de la seconde incluent l'attribut "[EXTERNAL]".

Exemple 2.1 Mécanisme de partage des déclarations par attributs.

```
program    Calcul_Operation;
var operande1, operande2, resultat : [EXTERNAL] INTEGER;
[EXTERNAL] procedure Somme;
EXTERN;
[GLOBAL] procedure Graphique;
         begin
         end;
begin

...

read (operande1, operande2)
Somme;

write (resultat)

...

end.
```

◆

```
module Operation;
var operande1, operande2, resultat : [GLOBAL] INTEGER;
[EXTERNAL] procedure GRAPHIQUE;
EXTERN;
[GLOBAL] procedure Somme;

begin
         resultat : = operande1 + operande2;
         Graphique;
end;

end.
```

■

Le second mécanisme, mis à la disposition des programmeurs, permet aux unités de se partager des variables, des procédures, des fonctions, des constantes et des types grâce à l'utilisation de fichiers d'environnement. Un fichier d'environnement, créé à partir d'une unité compilable, consiste dans la description sous une forme manipulable par un compilateur des déclarations de cette unité. Le partage des déclarations se fait par un héritage de fichiers d'environnement. L'effet de cet héritage est d'incorporer les déclarations de l'unité générant l'environnement directement dans l'unité héritant de celui-ci.

Exemple 2.2 Mécanisme de partage des déclarations par environnement.

```
[ENVIRONNEMENT ('Déclarations')] module Decls;
var operande1, operande2, resultat : INTEGER;
end.

[INHERIT ('Declarations')]program Exemple;

begin

        resultat : = operande1 + operande2;

        ...

end.
```

■

Avec cette technique, seules des unités écrites en VAX/Pascal peuvent s'échanger des déclarations, mais le compilateur effectue une vérification de la concordance des types. Pour ces quelques raisons, nous y serons dans la suite du travail plus particulièrement attentifs.

2.2. Modèle du cycle de vie.

Le modèle du cycle de vie d'un logiciel qui va servir de base à la conception d'un schéma E/R va être présenté succinctement dans cette section. Il est issu du modèle de cycle de vie en cascade dont les grands principes ont été décrits à la Section 1.2.2.

La première phase envisagée dans notre modèle est la conception d'une architecture logique. Nous avons choisi de structurer un projet logiciel en un certain nombre de modules. Ils sont classiquement définis comme des unités de spécification, de conception détaillée, de validation et de modification.

La relation entre modules est typiquement la relation utilisée. Un module A utilise un module B si la validité du module A dépend de la disponibilité d'une version correcte du module B.

Cette relation est exprimée par l'intermédiaire d'un niveau de décomposition plus fin, l'objet. Un objet est une unité de décomposition élémentaire définie dans un module et susceptible de servir à d'autres modules. La technique de spécification choisie pour les objets est la spécification par assertions réduite à la définition des arguments, des résultats et de leurs types. La construction d'algorithmes en pseudo-code pour chaque module et objet n'a pas été prise en compte.

Exemple 2.3 Résultat de la conception de l'architecture d'un module.

```
module : calcul_résultat
objet utilisé :  /
objet défini :

        calcul_operation : argument : operande1 : INTEGER
                                      operande2 : INTEGER
                                      operateur  : CHAR
                           résultat : resultat : INTEGER
```

Dans cet exemple, le module "calcul_resultat" ne va utiliser aucun objet défini dans d'autres modules. Il définit un seul objet "calcul operation". Cet objet est spécifié par la liste des arguments, des résultats et de leurs types.
■

L'utilisation du langage modulaire VAX/Pascal dans la phase de codage permet de considérer l'implémentation d'un module comme la composition de deux parties qui sont l'interface du module et sa réalisation au corps. L'interface indique toutes les ressources du module qui sont les procédures, les fonctions, les variables et les types visibles à l'extérieur du module. Le corps est la partie qui réalise les mécanismes permettant à un module de satisfaire à ses spécifications. Tous les détails de réalisation sont dans le corps et demeurent cachés aux autres modules.

Les principaux avantages d'une séparation de l'interface d'un module de sa réalisation sont explicités dans (Belkhatir 88). Nous nous contenterons donc ici d'en donner une brève énumération :

- mécanisme d'abstraction : nous pouvons définir une interface indépendamment du corps associé.

- minimisation des étapes de recompilation des modules "clients" : une recompilation n'est nécessaire que dans le cas de la modification de l'interface utilisée.
- support favorable à la définition des types abstraits de données : un module peut cacher les détails de leur représentation.

Déjà présente dans la phase de conception, cette distinction va donc aussi se rencontrer dans la phase de codage de notre modèle. Elle consistera à produire deux textes source différents correspondant respectivement à l'interface et au corps.

Exemple 2.4 Distinction entre le texte interface et le texte corps.

```
module calcul_resultat_interface;
procedure calcul operation (operande1, operande2 : INTEGER.
                            operateur :  CHAR;
                            var resultat : INTEGER);
   begin
   end;

end.
```

Texte interface en VAX/Pascal du module calcul résultat.

```
module calcul_résultat_corps;

procedure calcul_operation (operande1, operande2 : INTEGER
                           opérateur :  CHAR;
                           INTEGER);
                           var resultat :
   begin
   end          ... texte de la procédure ...

end.
```

Texte corps en VAX/Pascal du module calcul_resultat.

■

DE RAEDEMAEKER

Besoins du logiciel
(terme employé dans plusieurs modèles !)

1

1.2.6 p.14 Conclusion : on ramène tout en un seul modèle.

p.20 quelle phrase !

p.28 fig. et les boucles ?

Dans un but de simplicité conceptuelle, nous n'avons pas voulu envisager une phase de tests dans notre modèle. Aucune méthode de validation des textes source, soit par des jeux de test pour établir la présence d'erreurs, soit par une vérification (preuve) établissant l'absence d'erreurs, soit par la "construction de programmes" corrects n'a donc été reprise ici.

En permettant l'application au texte source de la gestion de versions, nous tenons compte des diverses modifications que l'on peut leur apporter. Une phase de maintenance est, par là même, bien présente.

La Figure 2.1 représente le modèle de cycle de vie résultant de notre approche

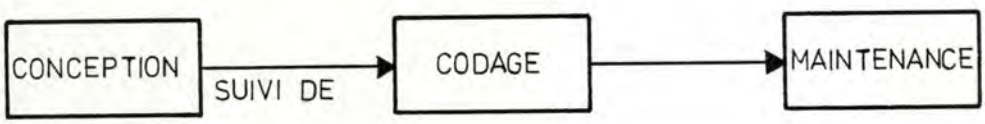


FIGURE 2.1 MODELE DU CYCLE DE VIE ETUDIE.

2.3. **Présentation générale du schéma E/R.**

Le schéma E/R, présenté dans cette section, va nous permettre de dégager tous les concepts pertinents au modèle décrit à la Section 2.2 ainsi que les relations entre ces différents concepts. Nous voudrions aussi que ce schéma rende compte des différentes étapes qui permettent d'obtenir une forme exécutable des textes source en VAX/Pascal.

Nous avons dégagé trois étapes principales dans le processus "architecture vers une forme exécutable" qui vont, chacune, donner lieu à un sous-schéma particulier.

- La première étape, développée dans un sous-schéma "architecture", met en évidence la notion de conception d'une architecture, c'est-à-dire le processus de structuration d'une application en divers composants et en relations bien définies entre ces composants.
- Une deuxième étape, donnant lieu à un sous-schéma "source", exprime l'organisation, les modifications et le stockage des textes source associés aux composants mis en évidence dans l'architecture logique. Cette étape intègre donc les notions développées dans les phases de codage et de maintenance de notre modèle.
- Enfin, une troisième étape, développée dans un sous-schéma "objet", exprime l'organisation et le stockage des éléments objet obtenus à partir des éléments source corespondants.

Ces trois sous-schémas vont maintenant être présentés.

2.3.1. Le sous-schéma "architecture".

Tout projet logiciel peut être structuré, selon la démarche présentée à la Section 2.2 , en un certain nombre de composants appelés modules. Un module est caractérisé par son nom et l'objectif qui lui est assigné. De plus, un module peut soit définir ou soit utiliser des objets. Un objet est une unité de décompostion élémentaire d'un module réalisant un traitement ou définissant une structure de données. Il

n'est défini que par un et un seul module mais peut être bien évidemment utilisé par plusieurs modules.

Face à cette distinction, nous pouvons spécialiser le concept d'objet par l'introduction de deux "sous-concepts" : les procédures-fonctions correspondant aux traitements et les types de données correspondant aux structures de données. Une procédure-fonction est caractérisée par son nom ainsi que le type de valeur qu'elle peut prendre; un type de données est caractérisé quant à lui par les propriétés inhérentes à sa structure.

Une procédure-fonction peut posséder dans sa déclaration des paramètres arguments ou résultats. Un paramètre est défini comme tout unité d'information pouvant être échangée avec des procédures ou des fonctions. Un paramètre peut être caractértisé par son nom ainsi que par le type de valeur qu'il prend; un paramètre peut appartenir à plusieurs procédures ou fonctions.

Le résultat unique de la transformation d'un module en vue de son exécution par une machine particulière dans un certain langage procédural est appelé implantation de ce module. Cette définition quoique très vague nous semblait pertinente à introduire ici pour expliquer les étapes suivantes de la réflexion. Une implantation est de par sa définition, dédicacée à un seul module.

Une implantation est toujours décomposée en deux parties qui sont l'interface et le corps. L'interface indique toutes les ressources du module visibles à l'extérieur du module. Le corps est la partie qui réalise les objectifs d'un module. Nous ferons remarquer que même si un module ne définit aucun objet le concept d'interface existe toujours pour ce module mais ne contient aucune indication de ressource, nous dirons qu'elle est vide.

La Figure 2.2 qui suit représente l'affinement E/R de la structure décrite dans cette section.

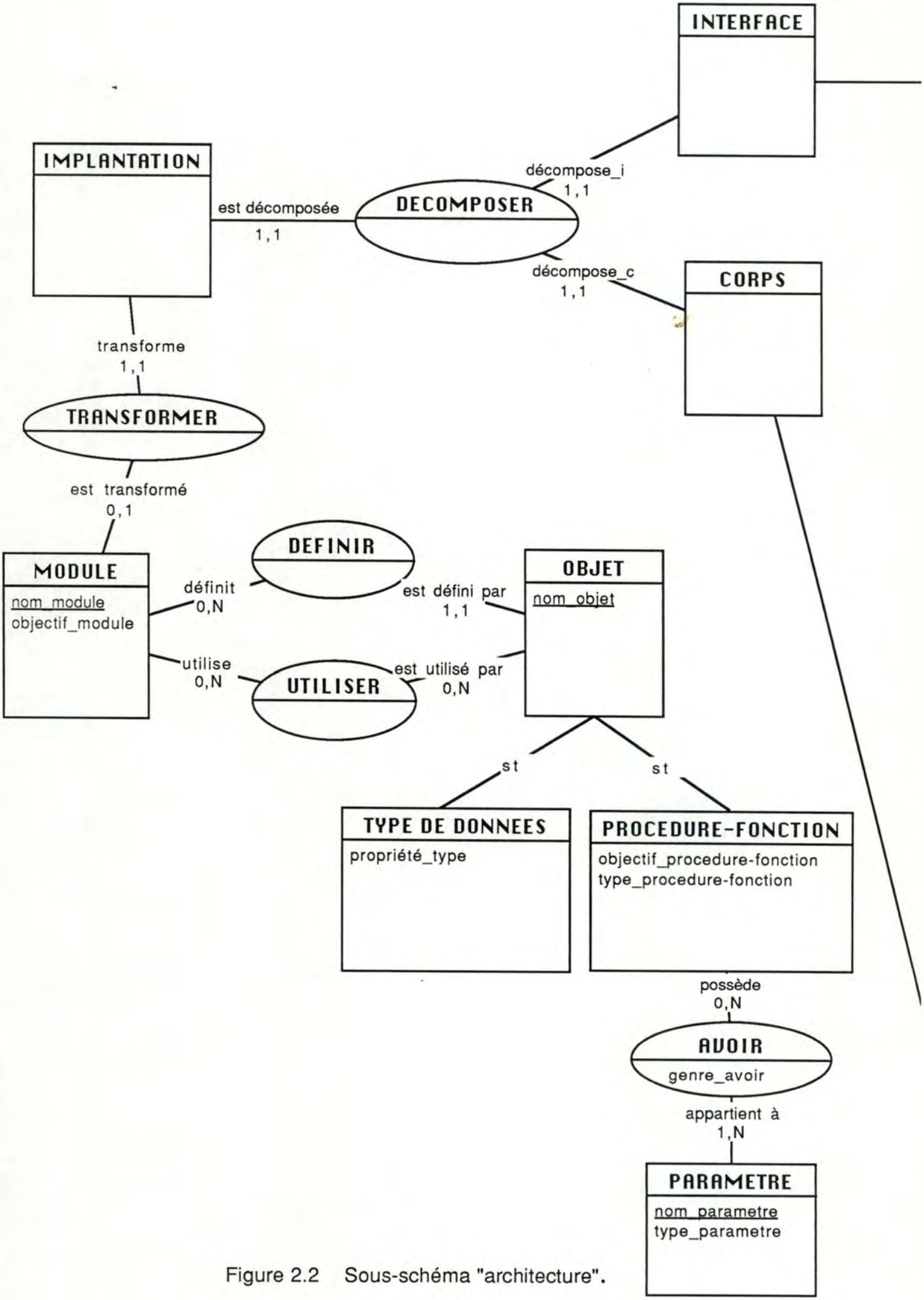


Figure 2.2 Sous-schéma "architecture".

2.3.2. Le sous-schéma "source".

Les concepts d'interface et de corps d'un module de l'architecture sont respectivement représentés dans un langage de programmation sous forme d'un ou plusieurs textes codés dans ce langage.
Nous les noterons respectivement par texte interface et texte corps. De plus, nous remarquerons qu'une interface vide n'est pas représentée par aucun texte source.

Un problème bien connu en gestion de configurations est celui de la double maintenance. Il résulte de la difficulté de garder identiques les multiples copies d'un même logiciel. Une solution à cette problématique est bien évidemment d'éviter les copies multiples.

Pour répondre à cette exigence, nous allons regrouper les définitions d'objets communes à un texte d'interface et au texte corps lui correspondant, dans un troisième, le texte partagé.
Pour être complet, le texte interface et le texte corps devant, en utilisant le mécanisme d'"include" de fichiers fournis par le langage VAX/Pascal, insérer ce texte partagé.
Un texte corps ne correspondant à aucun texte interface n'insère bien évidemment pas de texte partagé.

Exemple 2.5 Utilisation d'un texte partagé.

```
procedure calcul_operation (operande1, operande2 : INTEGER;
                  operateur : CHAR; var resultat : INTEGER);
forward; /*   convention propre au langage VAX/Pascal
permettant une défintion ultérieure des procédures -
                                                  fonctions */
```

Texte partagé en VAX/Pascal du module calcul_resultat.

◆

```
module calcul_résultat_interface;

 ...insertion du texte partagé...

procedure calcul-operation

begin

end;

end.
```

Texte interface en VAX/Pascal du module calcul_résultat.

◆

```
module calcul_résultat_corps

 ...insertion du texte partagé...

procedure calcul_operation;

     begin

        ...texte de la procédure ...

     end;

end.
```

Texte corps en VAX/Pascal du module calcul_résultat.

■

Les trois textes que nous venons de mettre en évidence possèdent tous les mêmes caractéristiques à savoir un nom, le langage de programmation dans lequel ils sont décrits et le texte correspondant. Cette remarque nous a donc amené à considérer un objet générique de plus haut niveau, l'élément source. Un élément source est donc défini comme étant tout texte séquentiel manipulable sous un format ASCII.

Nous allons tenir compte de l'évolution dans le temps de ces éléments source en développant les concepts de révision et d'alternative qui exprimeront les informations retraçant cette évolution. Cette distinction nous permet de tenir compte d'un des buts principaux de la gestion de configurations qui est le contrôle des modifications apportées à un logiciel.

Aucun élément source n'existe en une seule version mais évolue en de multiples versions. Nous distinguerons deux types d'évolutions, les révisions et les alternatives, faites en pratique.

Le concept de révision correspond à différentes formes d'évolution d'un élément source durant le développement et la maintenance comme par exemple le raffinement successif et les corrections d'erreur.
Une révision est souvent considérée comme une nouvelle version destinée à remplacer l'ancienne. Les révisions apparaissent dans un ordre linéaire, relatif à la séquence temporelle dans laquelle elles sont créées.

Le concept d'alternative correspond à des développements parallèles indépendants. Les éléments source gérés en alternative correspondent soit à des alternatives fonctionnelles ou d'implémentation, soit à des développements parallèles indépendants. Une alternative peut évidemment évoluer séquentiellement en révisions.

Enfin, nous allons envisager qu'un élément source est contenu dans une ou plusieurs librairies source. Une librairie source correspond à un lieu physique sur un dispositif informatique existant et indépendant, où il est possible de stocker des textes sous la forme ASCII.

La Figure 2.3 présente l'affinement E/R de la structure globale qui vient d'être décrite dans cette Section.

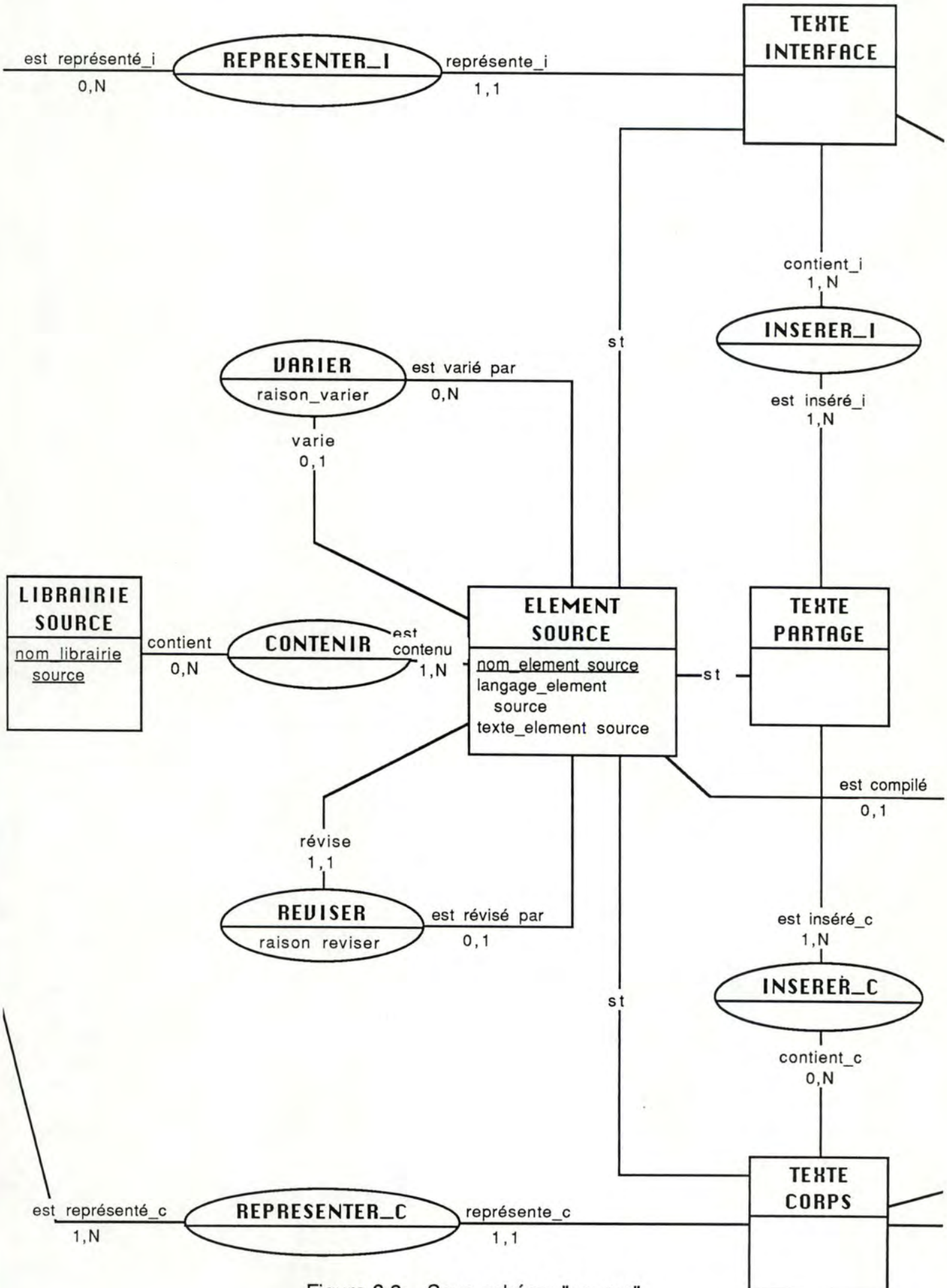


Figure 2.3 Sous-schéma "source".

2.3.3. Le sous-schéma "objet".

Les éléments source, mis en évidence dans la description du sous-schéma "source" peuvent être compilés à l'aide d'un compilateur approprié au langage de programmation dans lequel ils ont été décrits. Cette compilation génère, si elle réussit, un seul élément objet pour chaque élément source. Il est donc clair que, pour un élément objet, il ne correspond qu'un élément source.

Un élément source est défini comme étant un texte manipulable dans un format résultant d'une compilation. Il est caractérisé par un nom.
En particulier, un texte interface générera un environnement et un texte corps générera un élément relogeable. Remarquons que nous retrouvons au niveau des éléments objet la même notion de généralisation/spécialisation mise en évidence pour les éléments source.

Suite à la technique de partage des déclarations entre untés compilables déjà évoquée à la Section 2.1 , un texte corps peut hériter de plusieurs environnements créés par l'interface d'autres modules de l'architecture, selon que ce texte corps utilise ou non des objets définis dans d'autres modules. Un environnement peut être hérité par plusieurs textes source.

Un élément objet est reçu dans un ou plusieurs supports informatiques de nom générique pool objet permettant ainsi sa conservation.

Un pool objet est caractérisé par un nom et possède une existence propre, qu'il stocke ou non des éléments objets.

La Figure 2.4 présente un affinement E/R de cette structure globale.

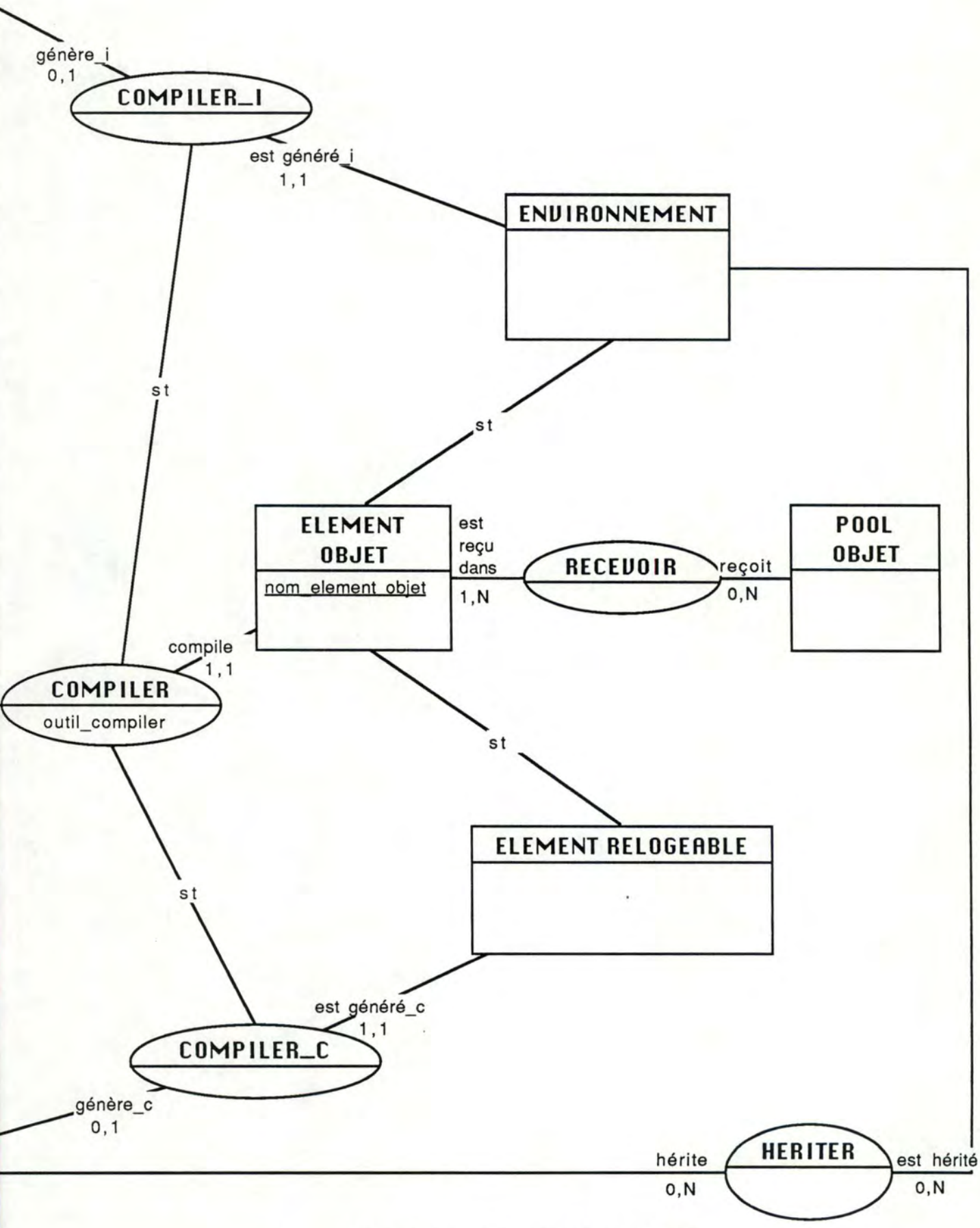


Figure 2.4 Sous-schéma "objet".

2.4. Spécification fonctionnelle de l'outil.

L'intérêt de cette section est de présenter les spécifications d'un outil logiciel intégré à une base d'objets et permettant une automatisation du début de la phase de codage de notre modèle (Section 2.2).

Plus précisément, notre programme utilitaire va, par la gestion (consultation et mise à jour) des informations de la base d'objet, générer, pour chaque module, les textes sources lui étant associés. Le texte interface sera dans sa version définitive tandis que le texte source sera dans une forme squelette formée des entêtes des procédures ou des fonctions, ou des définitions des types de données.

Les arguments et les résultats.

Le programme demande en arguments :

- les données issues des spécifications de tous les modules de l'architecture.

Les préconditions sur ces arguments sont :

- les données, provenant de l'architecture doivent être complètes et cohérentes.

Le programme produit comme résultats :

- pour chaque module, le nom et le code correspondant aux textes source, corps et partagé,
- des informations permettant de gérer d'une part la transformation d'un module en plusieurs textes source et d'autre part les éléments source entre eux.

Les postconditions sur ces résultats sont :

- Les textes source sont syntaxiquement corrects par rapport au langage VAX/Pascal,
- les informations de gestion sont complètes et cohérentes.

Exemple 2.6 Production automatique des textes VAX/Pascal source correspondant au module calcul-opération.

```
procedure calcul_operation
        ( operande_1   : INTEGER;
          operande_2   : INTEGER;
          operateur    : CHAR;
          var resultat  :  INTEGER
        );

  forward;
```

Texte partagé.

◆

```
module calcul_resultat_interface;
 ... insérer le texte partagé...
procedure calcul_operateur;

   begin

   end;

 end.
```

Texte interface.

◆

```
module calcul_resultat_corps;

 ... insérer le texte partagé ...

procedure calcul_operation

   begin
   end;
end.
```

Texte corps.

■

2.5. Solution.

Cette section va présenter, dans un premier temps, le processus de conversion des informations du schéma conceptuel décrit à la Section 2.3. en un ensemble de tables gérées par le Système de Gestion de Bases de Données (SGBD) RTI/INGRES, support à la base d'objets.

Dans un second temps, nous présenterons l'architecture de l'outil envisagé à la Section 2.4.

2.5.1. Mise en oeuvre de la structure de données.

La démarche, retenue dans notre travail, pour produire de façon systématique une base de données qui respecte la spécification conceptuelle est complètement explicitée dans (Hainaut 86). Nous n'en rappellerons donc ici que les grands principes.

Le schéma E/R a tout d'abord, été transformé, selon les concepts du Modèle d'Accès Généralisé (MAG), en un schéma MAG des accès possibles. Ce dernier schéma, produit systématiquement, peut être trouvé en Annexe 1.

Un schéma MAG est dit conforme à un SGBD, si les constructions de ce schéma obéissent à la spécification du SGBD. Nous avons donc rendu le schéma MAG conforme au modèle relationnel. Il est présenté en Annexe 2.

Pour ce faire, chaque type d'entité est normalement transformé en une table du modèle relationnel. Les types d'attribut deviennent des types d'item et l'identifiant du

type d'entité devient l'identifiant de la table.

A chaque type d'association au moins binaire sans attribut, à chaque type d'association avec attribut et à chaque type d'association binaire sans attribut plusieurs-à-plusieurs (N-M) correspond une table dans la structure de données relationnelle. Les types d'association ayant des types d'attribut deviennent des types d'item de la table correspondante. De plus, les rôles des types d'entité en relation deviennent des types d'item.

Un type d'association un-à-un (1-1) ou un-à-plusieurs (1-N) disparaît dans le modèle relationnel. L'identifiant du type d'entité du côté "1" est incorporé à la table issue de l'autre type d'entité.

Pour nous permettre de générer les noms des textes source plus aisément, nous avons adjoint au type d'article "module" un second identifiant désigné par l'item "nom interne". Ce nom doit être introduit par le membre du projet chargé de la gestion de la structure de données.

Nous avons, de plus, regroupé les types d'entité "implantation", "corps", et "interface" dans une seule table "transformer" où un item genre permet de distinguer entre une interface et un corps.

Seules les tables couvrant le sous-schéma "architecture" et les tables couvrant le sous-schéma "source" ont été implémentées. Le format des colonnes d'une table ne relève pas d'une analyse précise et, par là même, est donc purement

arbitraire. Le choix de la structure de stockage des lignes d'une table ne fait pas non plus l'objet d'une réflexion particulière; est donc utilisé la structure heap, par défaut pour INGRES.

Tous les accès à la base de données seront réalisés grâce à un des deux langages d'interrogation supportés par le logiciel INGRES, le Structured Query Language (SQL) - le second étant QUEL. Sa description complète est présentée dans (INGRES/SQL). Les requêtes SQL permettent de trouver, de gérer et de modifier les données existantes dans une base de données relationnelle.

2.5.2. Brève description de l'architecture.

Dans son état de développement actuel, l'outil se compose de trois programmmes écrits en langage PASCAL : le programme partagé, le programme interface et le programme corps.

Une requête SQL permettant l'accès aux données de la base INGES peut être englobée dans un programme d'application en langage PASCAL par l'intermédiaire du Embedded SQL PASCAL. Une description complète des principes généraux du Embedded SQL (ESQL) dans un langage de programmation procédural se trouve dans (INGRES/ESQL). Plus particulièrement, le ESQL PASCAL est décrit dans (INGRES/PASCAL).

Le programme partagé construit les textes ayant pour suffixe "_ ins.pas". Le programme interface construit quant à lui les textes ayant pour suffixe "_ int.pas". Enfin, le programme corps construit les textes ayant pour suffixe "_ imp.pas". Ces programmes mettent à jour la base de données. Les programmes interface et corps utilisant de l'information créée par le programme partagé, celui-ci doit donc être exécuté en premier lieu.

2.6. Evaluation de l'outil présenté.

Nous tenons à signaler l'aide appréciable qu'un outil tel que celui qui vient d'être présenté dans ce chapitre peut apporter dans l'élaboration des premières versions des textes source. Il permet d'une part de soulager une partie du travail des programmeurs. D'autre part, il facilite d'éventuelles modifications de ces textes source.

Le principal désavantage de cet outil est qu'il est dédicacé à un langage partculier, le Pascal et même très spécialement, le VAX/Pascal. Ceci réflète, nous semble-t-il, dans le troisième sous-schéma (Section 2.3.3.) qui a été conçu selon les particularités du VAX/Pascal (la technique du partage des déclarations plus particulièrement). Les deux premiers sous-schémas (Section 2.3.1. & 2.3.2.) nous semblent par contre adéquats pour tout langage de programmation modulaire (ADA, Modula2).

De nombreuses critiques, nous en sommes conscients, peuvent encore être faites sur la démarche de construction elle-même. Elles sont principalement dues au cadre volontairement restreint de notre étude. Nous en signalerons quelques-unes, elles pourraient toutes faire l'objet d'évolutions ultérieures :

- étude approfondie des choix techniques de représentation des informations de la base de données,
- introduction de la notion de version d'objet dans l'outil.

Enfin, nous noterons que l'outil ne couvre qu'une phase particulière (le codage) du cycle de vie d'un logiciel. Il pourrait cependant servir dans la conception d'un atelier logiciel plus complet.

CHAPITRE 3 :

MODELE DE CYCLE DE VIE EN PROGRAMMATION LOGIQUE.

3.1. Introduction.

Comme pour la programmation impérative, la conception de programme logique a demandé la mise en oeuvre de méthodologie de développement. (Deville 89) propose une telle démarche.

La Section 3.2. de ce chapitre va proposer une modélisation selon les concepts d'Entité/Relation de cette méthodologie. Nous avons ajouté à ce modèle les notions permettant de prendre en compte une phase de maintenance et une phase de test.

La gestion de configurations peut être vue comme l'organisation, la documentation et le suivi de configurations de systèmes. Par système, nous entendons un objet qui consiste en de nombreuses parties, ce qui implique aussi bien les parties mécaniques et leurs assemblages, les systèmes électriques et leurs composants, que les systèmes logiciels et leurs modules.

Pour chacun des types de système ci-dessus, la définition de gestion de configurations n'a pas beaucoup changé depuis ces vingt dernières années. Les seuls changements se retrouvent dans les applications de la gestion de configurations et dans les supports automatisés fournis.

Une des plus vieilles et des meilleures définitions de configurations peut être trouvée dans le Standard Militaire du département de la défense américain (le DOD) publié en 1968 (DOD 68). Le standard définit la gestion de configuration comme "... une procédure de gestion qui inclut les composants suivants :

- Identification : l'identification des composants qui forment une version particulière du système,
- Contrôle : toutes les modifications du système doivent être contrôlées et identifiées,
- Statut : des informations explicitant toutes les modifications doivent être signalées et enregistrées,
- Vérification : le système doit être vérifié pour être conforme à la documentation de configuration".

Ce thème, considéré comme essentiel en ingéniérie matérielle, est bien défini en ingéniérie logicielle mais y est encore largement ignoré en pratique. La Section 3.3. essayera de spécifier un système de gestion de configurations des objets mis en évidence à la Section 3.2.

3.2. Modélisation du développement de programmes logiques.

3.2.1. Présentation générale de la méthodologie.

Nous allons donner tout d'abord un aperçu de la méthodologie de développement en programmation logique

présentée dans (Deville 89). Rappelons que le cadre de travail "logique" doit tenir compte de la puissance du langage de programmation cible, le PROLOG.

Le développement d'un programme PROLOG est décomposé en trois étapes significatives.

- La première étape requiert l'élaboration d'une spécification du problème à résoudre. Pour ce faire, aucun langage de spécification particulier n'est recommandé, une spécification peut être ainsi écrite en langage naturel. Elle contient en particulier une description de relation.
- La seconde étape consiste à construire une description logique sur base d'une spécification. Une description logique est une formule de la logique des prédicats du premier ordre. Dans ce sens, nous pouvons affirmer qu'elle est indépendante de toute sémantique procédurale, donc de tout langage de programmation. Nous signalons qu'une description logique peut subir certaines transformations afin de permettre la dérivation (voir étape prochaine) de procédures plus efficaces.
- La troisième étape permet de dériver, à partir d'une description logique, un programme logique (en PROLOG). Dans un premier temps, une description logique est traduite sous forme de clauses, la procédure logique pure. Cette procédure est, dans un second temps, optimisée pour produire un code PROLOG efficace.

La Figure 3.1 suivante exprime les relations entre les étapes que nous venons de décrire brièvement.

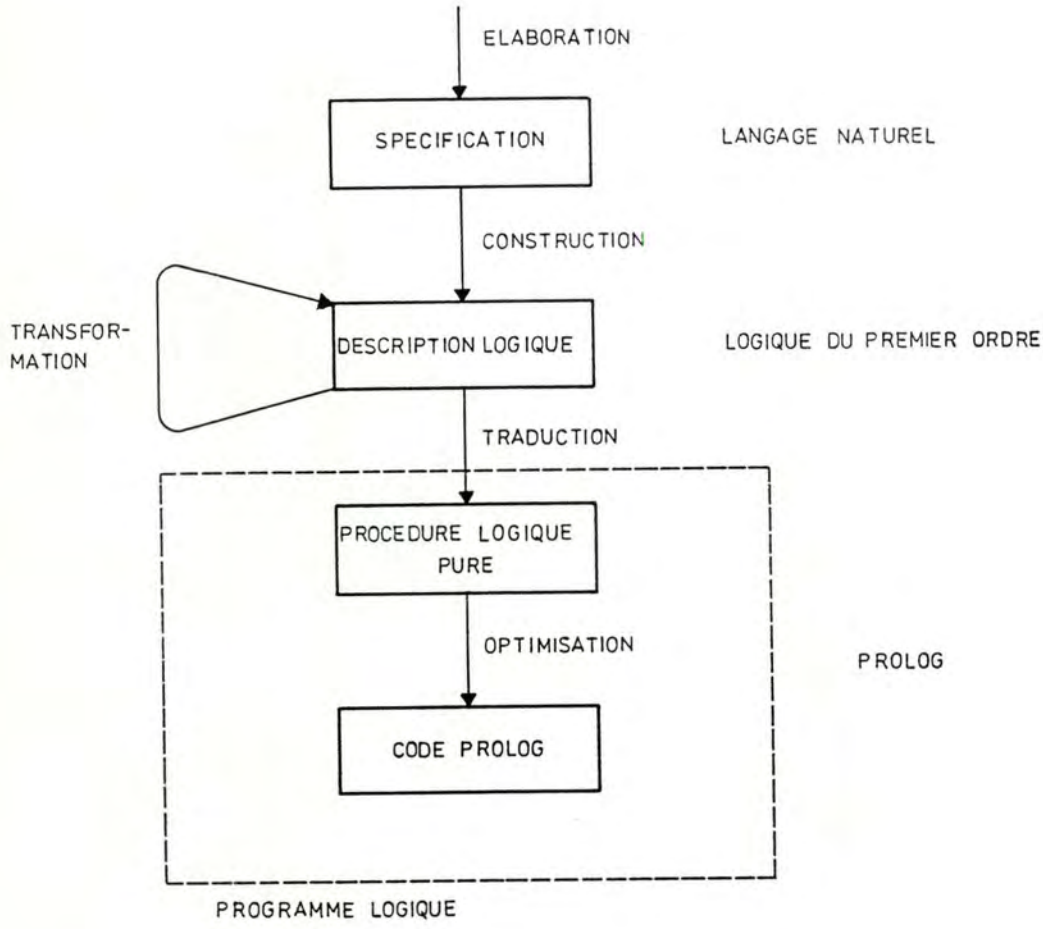


FIGURE 3.1 METHODOLOGIE DE DEVELOPPEMENT EN PROGRAMMATION LOGIQUE.

Nous voudrions évidemment que la modélisation de cette méthodologie s'articule autour de ces trois étapes principales. Aussi, nous est-il paru clair de représenter le résultat de chaque étape par un type d'entité et le processus menant d'une étape à une autre par un type d' association, Figure 3.2

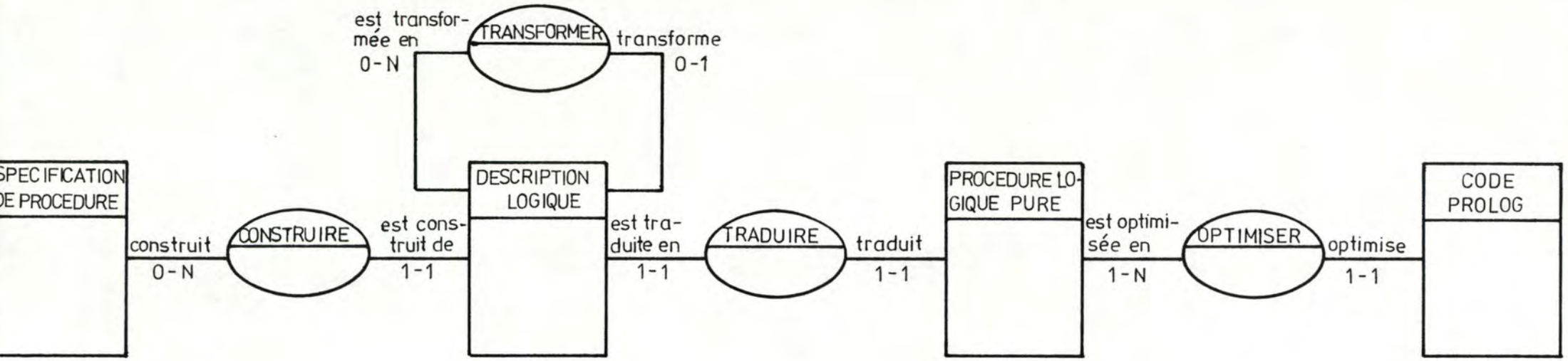


FIGURE 3.2 SCHEMA ARTICULANT LES ETAPES PRINCIPALES DE LA METHODOLOGIE PROLOG

Une spécification peut donner lieu à la construction d'une description logique. Nous ferons remarquer que ce procédé n'est pas unique. Afin de permettre une dérivation d'un programme plus efficace, une description logíque peut être transformée en une autre description . Le procédé de traduction étant déterministe ne produit donc à partir d'une description logique qu'une seule procédure logique pure.
Une procédure est toujours optimisée en code PROLOG mais peut en fournir plusieurs selon la technique d'optimisation utilisée.

3.2.2. Raffinement de l'étape "spécification de procédure".

La première étape du développement de programmes en PROLOG consiste donc dans l'élaboration de la spécification du problème. Comme nous l'avions déjà fait remarquer, aucun langage de spécification n'est imposé. Nous entendons par spécification, non pas l'énoncé des fonctions que le logiciel doit réaliser, ce qui correspond à la spécification des besoins, mais la spécification d'une procédure résolvant le problème, c'est-à-dire la phase de conception.

Nous avons dégagé comme caractéristiques pour une spécification de procédure :

- un nom identifiant la procédure,
- le nombre de paramètre de la procédure, c'est-à-dire son arité,
- une description des relations qui peuvent exister entre certains paramètres avant l'exécution de la procédure appelées restrictions sur les paramètres,

- une description de la relation qui existe entre les paramètres, appelée simplement relation,
- une description des utilisations possibles de la procédure, appelée la directionnalité de la procédure,
- une description des préconditions qui doivent être satisfaites par l'environnement de la procédure avant son exécution,
- une description des effets de bord (entrées-sorties,...)

Typiquement, chacune de ces caractéristiques peut être considérée comme un type d'attribut du type d'entité "spécification de procédure". Il convient pourtant de nuancer cette première idée en discutant plusieurs de ces caractéristiques.

Conformément à la logique, nous allons identifier une spécification de procédure par son nom mais aussi par le nombre de paramètres de cette procédure. Ces deux caractéristiques, le nom et l'arité, vont donc rester des types d'attribut.

A chaque paramètre de la procédure est assigné un type de valeurs. Nous pourrions gérer ces informations comme un type d'attribut décomposable et répétitif. Au vu de la méthodologie de développement de programme PROLOG, il nous a semblé plus pertinent de définir deux nouveaux types d'entité, à savoir; "paramètre" et "type".

Nous avons donc qu'une spécification de procédure possède un certain nombre de paramètres formels, égal à

l'arité de la spécification. Il est aussi clair qu'un paramètre peut appartenir à un nombre quelconque de spécifications. Cette situation est modélisée par la Figure 3.3 suivante.

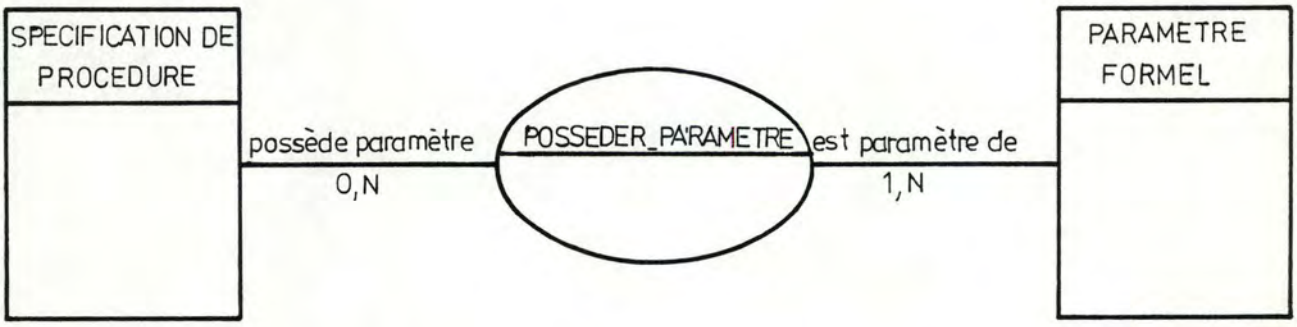


FIGURE 3.3 LES PARAMETRES D'UNE SPECIFICATION DE PROCEDURE.

Contrainte :

- la cardinalité maximale du rôle "possède paramètre" est égale à l'arité de la spécification de procédure correspondante.

Comme caractéristiques pour un paramètre formel, nous trouvons :

- le nom du paramètre formel,
- la place que le paramètre occupe dans la liste des paramètres d'une procédure, appelée la position du paramètre.

Nous prendrons comme identifiant d'un paramètre, son nom et l'identifiant de la spécification de procédure. Les caractéristiques deviennent des types d'attribut comme le montre la Figure 3.4

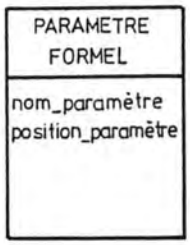


FIGURE 3.4. PARAMETRE FORMEL.

Contrainte :

- la valeur de l'attribut "position_paramètre" est inférieure ou égale à l'arité de la spécification de procédure.

Un paramètre formel a un et un seul type. Un type est carctérisé par :

- un nom identifiant le type.

Pour être complet par rapport à la méthodologie, nous avons envisagé d'inclure dans notre modèle une relation de sous-typage multiple. Ces informations se trouvent dans la figure 3.5

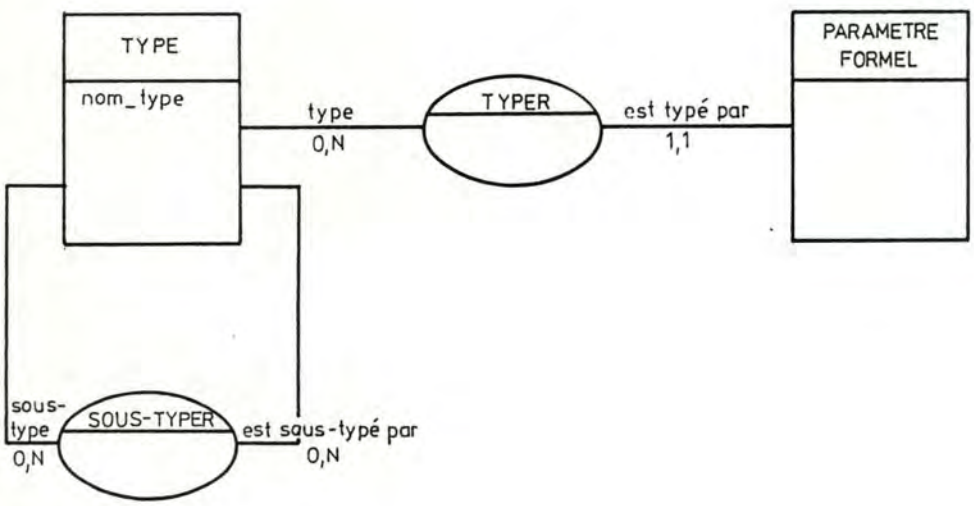


FIGURE 3.5. LE TYPAGE.

Pour des raisons méthodologiques qui seront explicitées à la Section 3.2.3. nous dirons qu'uun type de paramètre définit un certain nombre de relations bien fondées. Une relation bien fondée est considérée par un nom permettant de l'identifier, ces quelques remarques donnent le schéma de la Figure 3.6

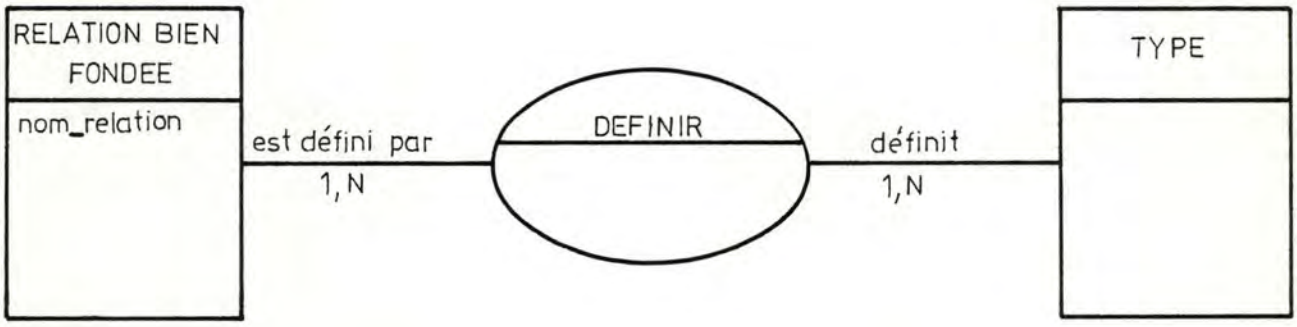


FIGURE 3.6 RELATION BIEN FONDEE.

Les relations entre certains paramètres et la relation unissant tous les paramètres d'une spécification de procédure sont des textes en langage naturel.

Ces carctéristiques seront donc gérées, au niveau de notre modèle, comme des types d'attribut.

La partie décrivant les directionnalités d'une spécification correspond à la description des utilisations possibles de la procédure. Une directionnalité décrit la forme des paramètres réels avant et après l'exécution de la procédure. Pour chaque paramètre, trois formes ont été retenues : instancié, variable, ni instancié, ni variable.

Une directionnalité est notée

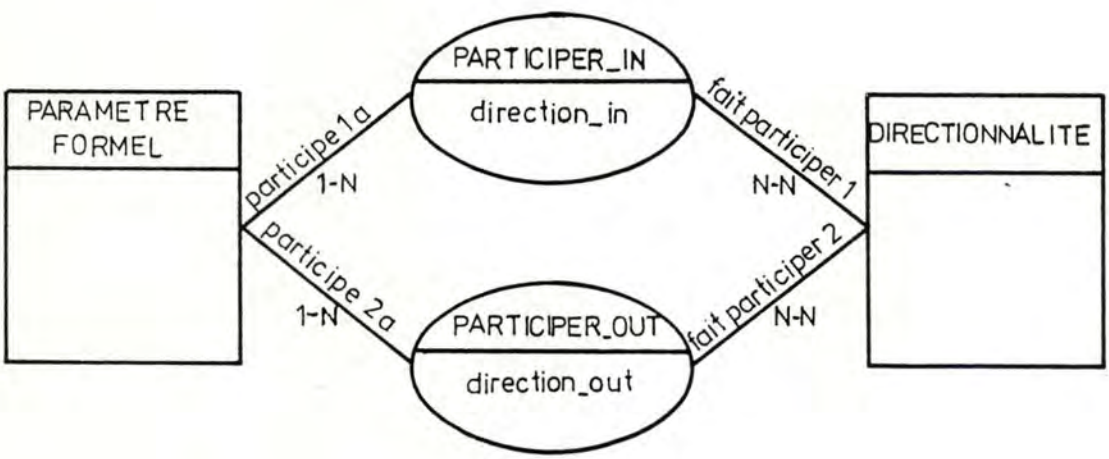
in (m1, m2,...,mn) : out (M1, M2,...,Mn)

où n correspond à l'arité de la procédure,

mi, Mi $\subseteq$ {instancié, variable, ni instancié ni variable}

La directionnalité d'une procédure est donc une liste non vide de Nd directionnalités.

Disposant d'un type d'entité "paramètre", il nous est possible de considérer les directionnalités d'une procédure autrement que comme un simple type d'attribut de type texte. En effet, un paramètre participe une seule fois dans la partie "in" et dans la partie "out" d'une directionnalité. une telle situation peut être modélisée de la façon suivante :



Il est clair qu'un tel schéma peut être fusionné favorablement par le schéma présenté à la Figure 3.7 suivante.

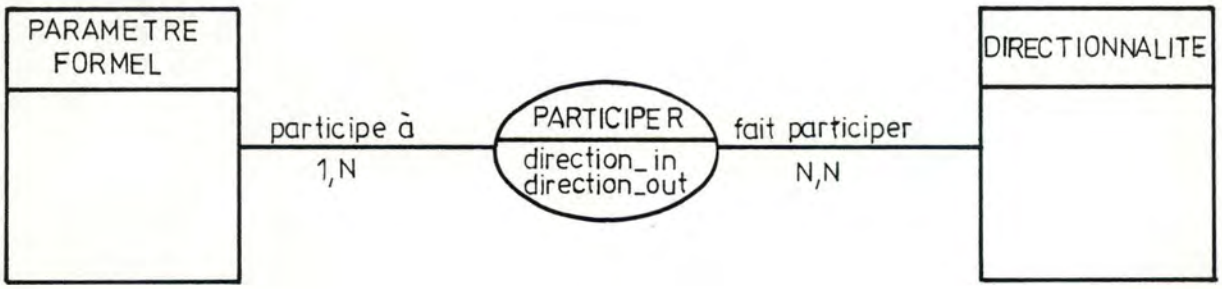


FIGURE 3.7 LES PARAMETRES DANS LES DIRECTIONNALITES.

Contraintes :

- la cardinalité maximale du rôle "participe à" est égale au nombre de directionnalités de la spécification de procédure correspondante.
- la cardinalité minimale du rôle "fait participer" sont égales à l'arité de la procédure correspondante.

●

Bien qu'il soit redondant pour le schéma, nous avons voulu gérer plus explicitement le lien existant entre une procédure et sa liste de directionnalités.
Il suit donc le schéma de la Figure 3.8

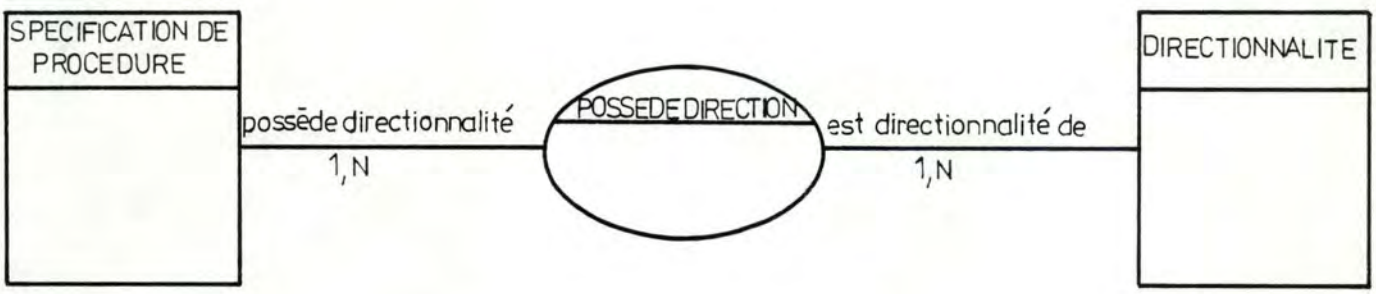


FIGURE 3.8. LA DIRECTIONNALITE D'UNE SPECIFICATION DE PROCEDURE.

A chaque directionnalité sont attachées des informations, au moyen d'un couple <Min-Max>, sur la multiplicité c'est-à-dire sur le nombre de substitutions résultant d'un appel à une procédure. Min et Max sont respectivement la borne inférieure et la borne supérieure du nombre de substitutions pour une directionnalité. nous pouvons considérer ces informations comme les caractéristiques d'une directionnalité. Ce qui donne une modélisation la Figure 3.9 suivante.

FIGURE 3.9 DIRECTIONNALITE.

Les descriptions des préconditions et des effets de bord sont, eux aussi, des textes en langage naturel et seront donc considérés comme de simples types d'attribut du type d'entité "spécification de procédure". Enfin, nous avons ajouté un type d'attribut "primitive" qui indique si la procédure spécifiée doit être considérée comme une procédure logique primitive ou non. Son intérêt devrait être mis en lumière dans les évolutions futures de la méthodologie. La Figure 3.10 résume les types d'attribut retenus pour le type d'entité "spécification de procédure".

FIGURE 3.10 SPECIFICATION DE PROCEDURE.

3.2.3. Raffinement de l'étape "description logique".

La deuxième étape de la méthodologie proposée par le développement de programmes en PROLOG est la construction d'une description logique à partir d'une spécification de procédure. Le procédé de construction est basé uniquement sur la sémantique déclarative de la logique et est complètement indépendant de la logique sémantique procédurale et du langage de programmation cible. Pour la construction, il n'est utilisé que les types des paramètres et les relations.

Une description logique est une formule bien fermée de la forme :

$$(\forall X1 \ldots \forall Xn)\ (p(X1, \ldots, Xn) \Longleftrightarrow (\exists Y1 \ldots \exists Ym)\ F)$$

où $n, m \geq 0$,

F est une formule de la logique du premier ordre sans quantificateur,

X1..Xn sont des variables,

Y1..Ym sont des variables,

p est un symbole de prédicat.

Les deux seules caractéristiques identifiées pour une description logique sont donc :

- un nom identifiant la description logique,
- le texte de la formule logique correspondant à la description logique, appelé le code de description.

Ces deux caractéristiques ne faisant l'objet d'aucune discussion, vont être considérées comme des types d'attribut du type d'entité "description logique" comme le montre la Figure 3.11

FIGURE 3.11 DESCRIPTION LOGIQUE.

La construction d'une description logique est certainement une des tâches les plus difficiles de la méthodologie proposée. Il n'existe pas de recette miracle pour résoudre cette tâche de création mais on peut toutefois présenter une méthode de construction générale. Cette méthode, qui ne convient qu'aux problèmes demandant une solution récursive, est basée sur les principes d'induction structurelle et de généralisation. Le principe d'induction structurelle correspond à une induction sur la structure des paramètres. Les autres problèmmes vont être résolus de manière directe.

L'objectif du processus de développement est d'obtenir, à partir d'une spécification de procédure, une description logique correcte. il peut être résumé par les 4 étapes suivantes :

- choisir un paramètre, appelé le paramètre d'induction,
- choisir une relation bien fondée sur le type du paramètre d'induction,
- construire les formes structurelles du paramètre d'induction, c'est-à-dire les cas posibles, dont le cas minimal du paramètre d'induction,
- construire les cas structurels, un cas structurel satisfait la relation pour une forme structurelle du paramètre d'induction.

Soient un ensemble E et $<$ une relation binaire définie sur E. Une séquence $x_1, \ldots, x_i, \ldots$ d'éléments de E est dite une séquence décroissante ssi $x_1 > \ldots > x_i > \ldots$

On dit que la relation binaire est bien fondée sur un ensemble E encore que (E,<) est un ensemble bien fondé si il n'existe pas de séquence décroissante infinie d'éléments de E.

Exemple Relation bien fondée.

La relation "plus petit que" est une relation bien fondée sur N, l'ensemble des entiers positifs mais pas sur l'ensemble des entiers positifs et négatifs.

■

Enfin, un élément e de E est un élément minimal de E (E,<) si il n'existe pas de e'de E tel que e'< e

L'aspect créatif est le plus important de la construction d'une description réside clairement dans le choix du paramètre d'induction et de la relation bien fondée. Les deux dernières étapes sont directement déterminées par les deux choix précédents.

Nous retiendrons donc ces deux choix comme caractéristiques principales du processus de construction. Elles deviendront, dans la modélisation en cours, des types d'attribut du type d'association "construire" comme représenté à la Figure 3.12. suivante.

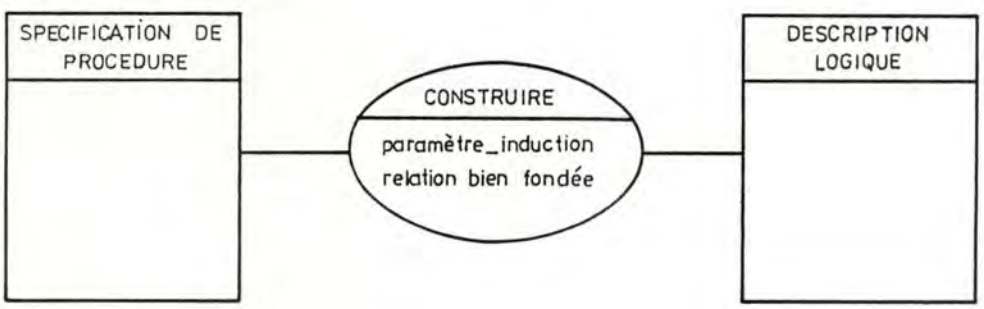


FIGURE 3.12 CONSTRUCTION D'UNE DESCRIPTION LOGIQUE

Contraintes :

- le nom du paramètre d'induction doit être dans l'ensemble des noms des paramètres formels de la spéciification de procédure correspondant à cette description,
- la relation bien formée doit être dans l'ensemble des relations bien fondées définies par le type du paramètre d'induction choisi.

Une fois une description logique construite, elle peut être transformée tout en préservant sa correction, en une autre description.

Une telle approche a de nombreux avantages. Elle permet de concentrer les efforts de l'analyste uniquement sur la construction d'une description logique correcte. Il est plus simple d'optimiser une description correcte que de corriger une description optimisée. Il existe plusieurs mécanismes de transformation, tous déterministes.

Nous n'avons pas relevé de caractéristique pertinente à une transformation.

En raison d'une évolution possible de la méthodologie, il nous a été demandé d'envisager la possibilité du prototypage d'une description logique. Nous aurons comme cactéristiques pour un prototype

- un nom identifiant le prototype,
- le texte du prototype, appelé le code du prototype.

Ces informations sont modélisées dans le schéma de la Figure 3.13 suivante.

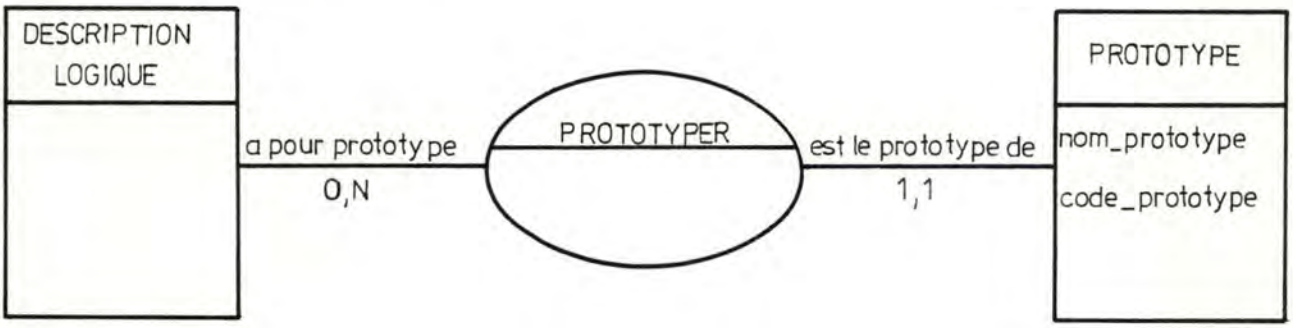


FIGURE 3.13. LE PROTOTYPAGE

Lorsque, pour un paramètre d'induction et une relation bien fondée, il n'est pas possible de construire une description de procédure à partir d'une spécification, nous pouvons généraliser cette spécification. Une spécification généralisée peut être plus simple à considérer dans le processus de construction.

Deux stratégies de généralisation d'une spécification de procédure sont présentées dans la méthodologie de programmation PROLOG. La première stratégie consiste à généraliser la structure d'un paramètre, elle est appelée la généralisation structurelle. En particulier, la généralisation sur un paramètre de type liste est appelée la généralisation tupling. La seconde stratégie caractérise un état général de calcul en déterminant ce qui déjà été fait et ce qui reste à faire. Cette stratégie, la généralisation de calcul, distingue la généralisation ascendante et decendante.

Une discussion avec le concepteur de la méthodologie nous a permis de mettre en évidence les caractéristiques des généralisations. Pour de plus amples informations, le lecteur se référera à (Deville 89). Pour chaque type de généralisation, nous avons le choix d'un paramètre d'induction. De plus, pour une généralisation structurelle, nous avons une information sur le type de paramètre d'induction appelé structure. Une généralisation ascendante est aussi caractérisé par des informations de typage, notée type_info_préf. Enfin, une généralisation descendante est caractérisé par un opérateur de composition.

Ces trois types de généralisation peuvent donner lieu à trois types d'association. Dans un but de clarté, nous les avons pourtant fusionnés comme le montre la Figure 3.14 suivante.

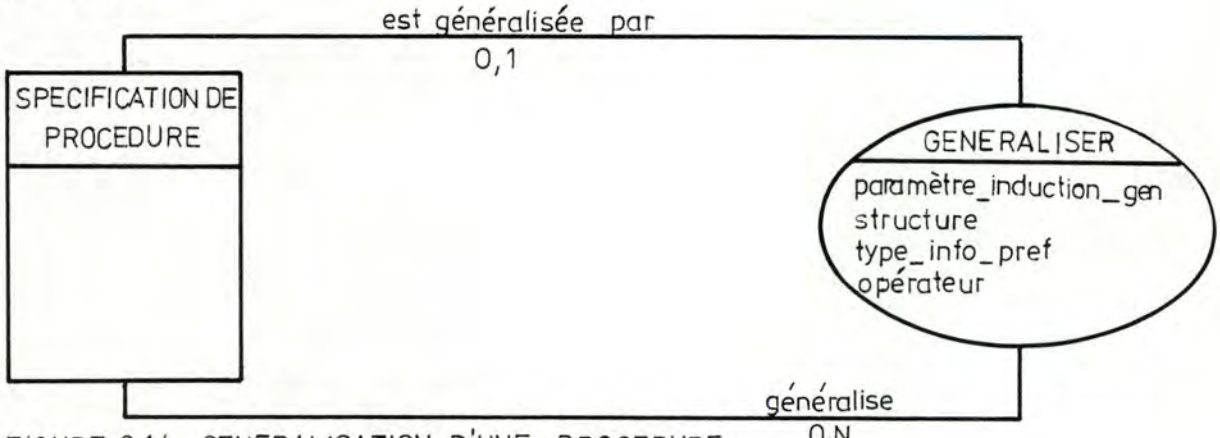


FIGURE 3.14. GENERALISATION D'UNE PROCEDURE

3.2.4. Raffinement de l'étape "procédure logique pure".

La troisième étape de la méthodologie proposée est la traduction d'une description logique en une procédure pure. Cette étape tient compte de la sémantique procédurale du langage de programmation cible. Les conditions d'application,

comme par exemple les directionnalités doivent être prises en compte. Le PROLOG est choisi comme langage de programmation cible.

La traduction qui peut être automatisée, correspond à une simple réécriture de la description logique sous forme de clauses logiques pures. Pour cette raison, nous n'avons relevé aucune caractéristique propre à la traduction. En ce qui concerne la procédure logique pure, elle est carctérisée par :

- un nom identifiant la procédure logique pure,
- le texte de la séquence de clauses logiques correspondant à la procédure logique encore appelé le code de la procédure.

Nous trouvons la modélisation de ces caractéristiques sous forme de types d'attribut à la Figure 3.15 suivante.

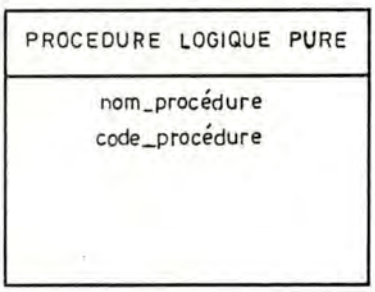


FIGURE 3.15. PROCEDURE LOGIQUE PURE.

Afin de satisfaire à un maximum de directionnalités définies dans la spécification de procédure correspondante, on peut dériver une procédure logique en une autre.
Le processus de dérivation correspond par exemple à la permutation entre clauses, à la permutation dans le corps des clauses, à l'ajout de littéraux effectuant des vérifications de type et de littéraux réalisant les entrées/sorties.

Dans le processus de dérivation, nous n'avons pas non plus relevé des caractéristiques propres au processus. Nous ferons simplement remarquer deux choses, le nombre de directionnalités satisfaites par une procédure logique est inférieur ou égal au nombre Nd de directionnalités définies dans la spécification correspondant à cette procédure. La méthodologie montre, qu'au maximum, Nd sont nécessaires pour que toutes les directionnalités soient satisfaites par au moins une procédure logique dérivée.

Les quelques constatations que nous venons de mettre en évidence sont modélisées par la Figure 3.16. suivante.

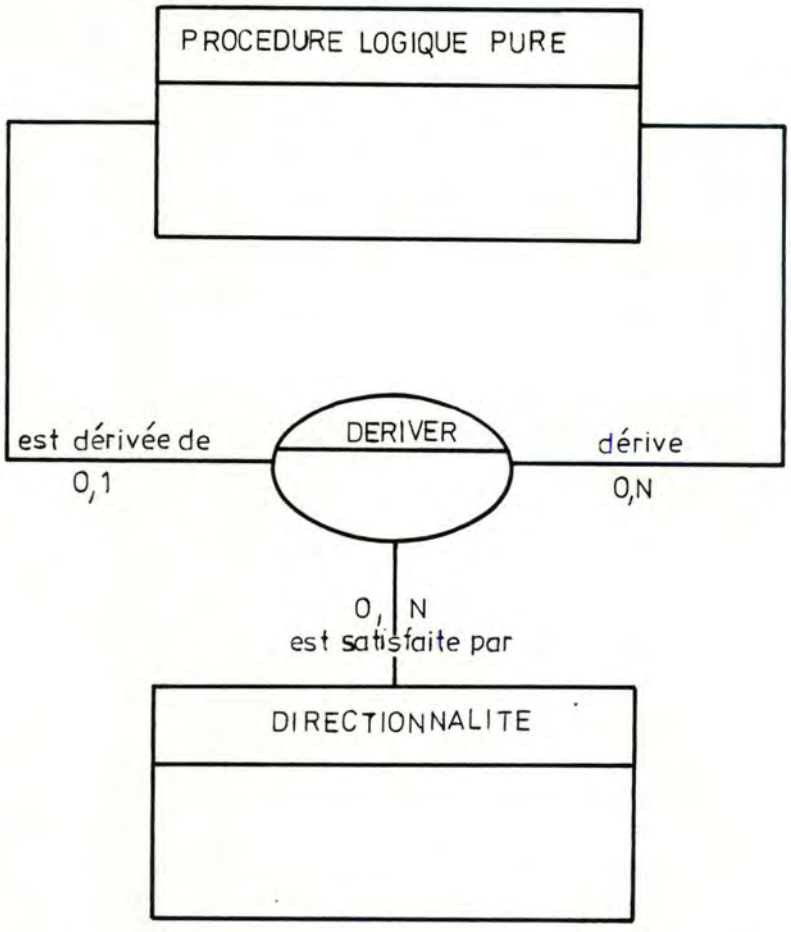


FIGURE 3.16. DERIVATION D'UNE PROCEDURE LOGIQUE PURE

Contraintes :

- la cardinalité maximale du rôle "dérive" est inférieure ou égale au nombre de directionnalités de la spécification de procédure correspondante.
- la cardinalité maximale du rôle "est satisfaite par" est inférieure ou égale au nombre de directionnalités de la spécification de procédure correspondante.
- la dérivation s'effectue toujours sur une procédure logique qui n'a pas été dérivée.

3.2.5. Raffinement de l'étape "code PROLOG".

La dernière étape du développement de programme logique est l'optimisation d'une procédure logique en un code PROLOG efficace. Le processus d'optimisation, par l'introduction de prédicats méta-logiques tels que cut et fail, assure un contrôle de l'exécution de la procédure.

Nous avons identifié comme caractéristiques pour un code PROLOG

- le nom permettant d'identifier un code PROLOG
- le texte du code PROLOG

Nous retrouvons ces caractéristiques comme types d'attribut du type d'entité "code PROLOG" comme le montre la Figure 3.17

FIGURE 3.7 CODE PROLOG

3.2.6. Extension du modèle E/R.

Outre le modèle E/R de la méthodologie de développement, le but de la Section 3.2. est d'une part d'introduire les concepts permettant de gérer les objets enversions multiples. D'autre part, nous étendrons la méthodologie avec la prise en compte dans notre modèle E/R de la notion de test permettant d'établir la présence d'erreurs dans une procédure.

3.2.6.1. Extension aux versions.

Nou rappelons que par version d'objet, nous entendons sa révision et son alternative. Une révision correspond à différentes formes d'évolution de l'objet; une alternative correspond à des développements parallèles indépendants d'un objet. Cette distinction peut être vue comme l'introduction dans la méthodologie d'une phase de maintenance.

Une première réflexion avec les utilisateurs nous a conduit à identifier les objets susceptibles de nécessiter

les versions. Ils sont au nombre de quatre à savoir la spécification de procédure, la description logique, la procédure logique pure et enfin le code PROLOG.

Nous obtenons donc qu'un objet peut être révisé par un autre et qu'un objet peut être alterné par plusieurs autres. Nous développerons dans les Sections 3.3.1. & 3.3.3. les caractéristiques d'une version, nous ne ferons que les signaler ici :

- une représentation des différences entre objets, appelés deltas,
- un texte explicitant les modifications effectuées sur l'objet, appelé la raison.

Ces constatations peuvent se modéliser comme montré à la Figure 3.18

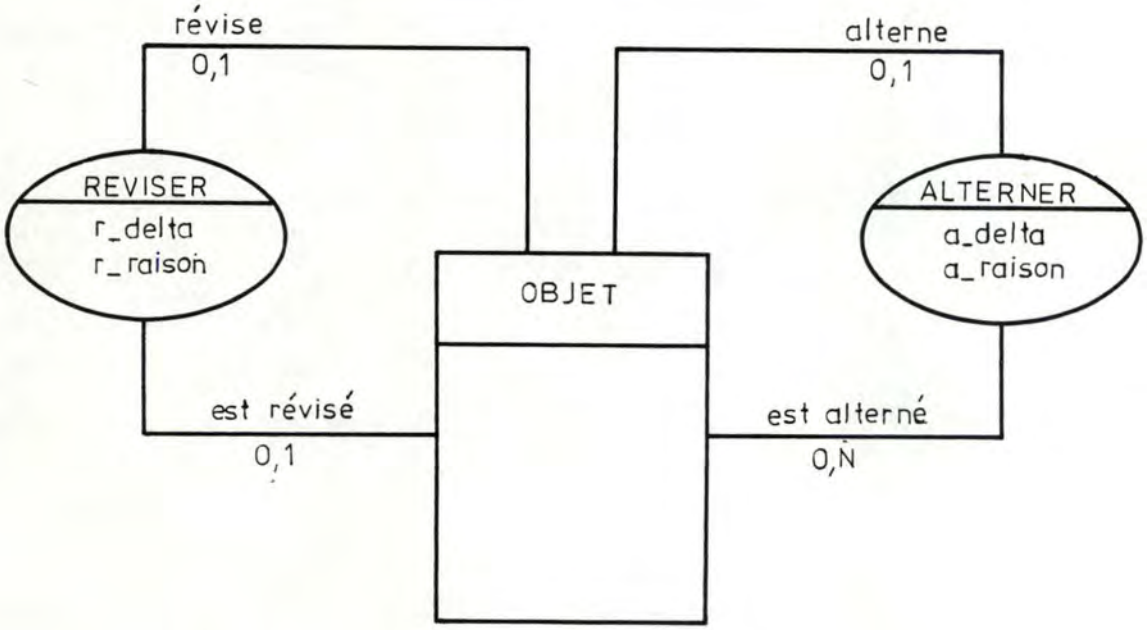


FIGURE 3.18. SCHEMA DE GESTION DES VERSIONS D'UN OBJET.

3.2.6.2. **Extension aux jeux de test.**

Nous voudrions introduire dans notre modèle E/R de la méthodologie la notion de test. Pour cela, la première chose est de se demander quel objet doit être testé. dans la réalité, ce sont les programmes qui subissent les tests, dans notre étude les codes PROLOG donc. Cependant, chronologiquement, les jeux de test sont conçus à partir de spécifications. De plus, nous soulignerons que, pour une signification, peut correspondre plusieurs programmes. Il s'en suit donc que le seul objet réquérant une phase de test est la spécification de procédure.

Donc une spécification peut posséder un certain nombre de jeux de tests. de par la définition de la directionnalité d'une procédure, il est aussi évident qu'un jeu de test doit satisfaire à au moins une directionnalité. Un jeu de test est caractérisé par

- un nom identifiant le test,
- un exemple de valeur de test.

Ceci se retrouve dans la Figure 3.19 suivante :

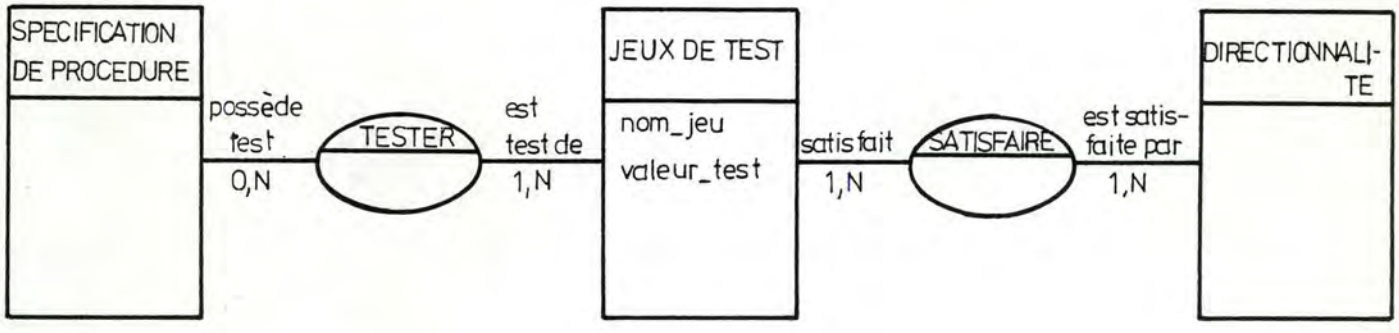
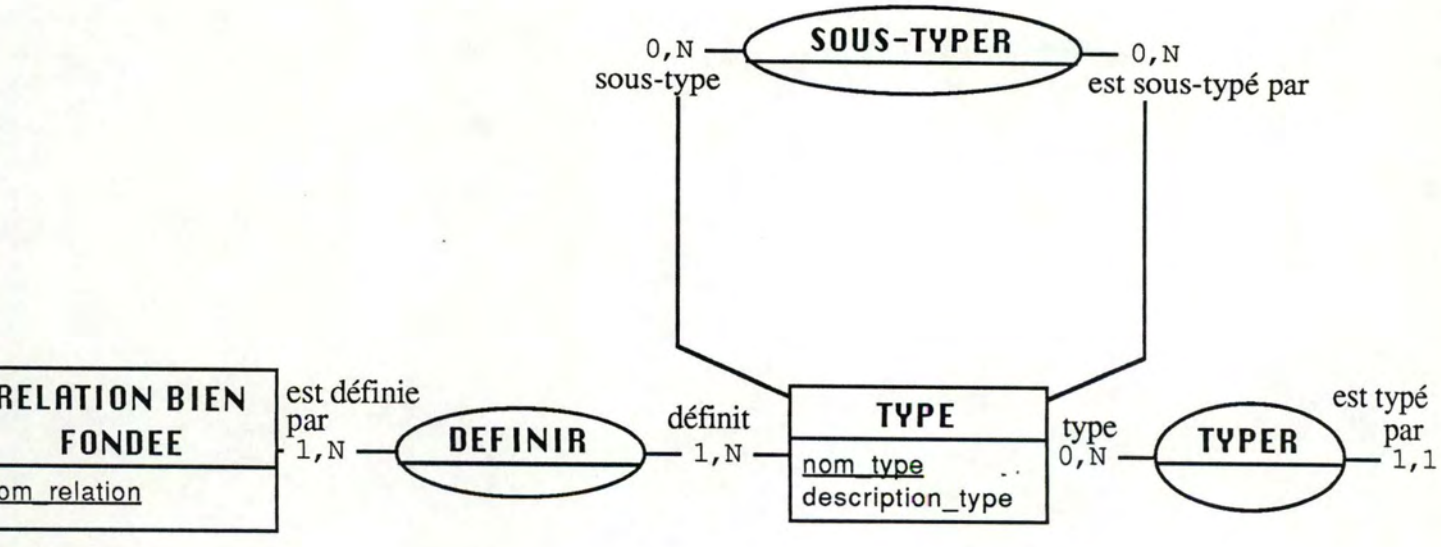


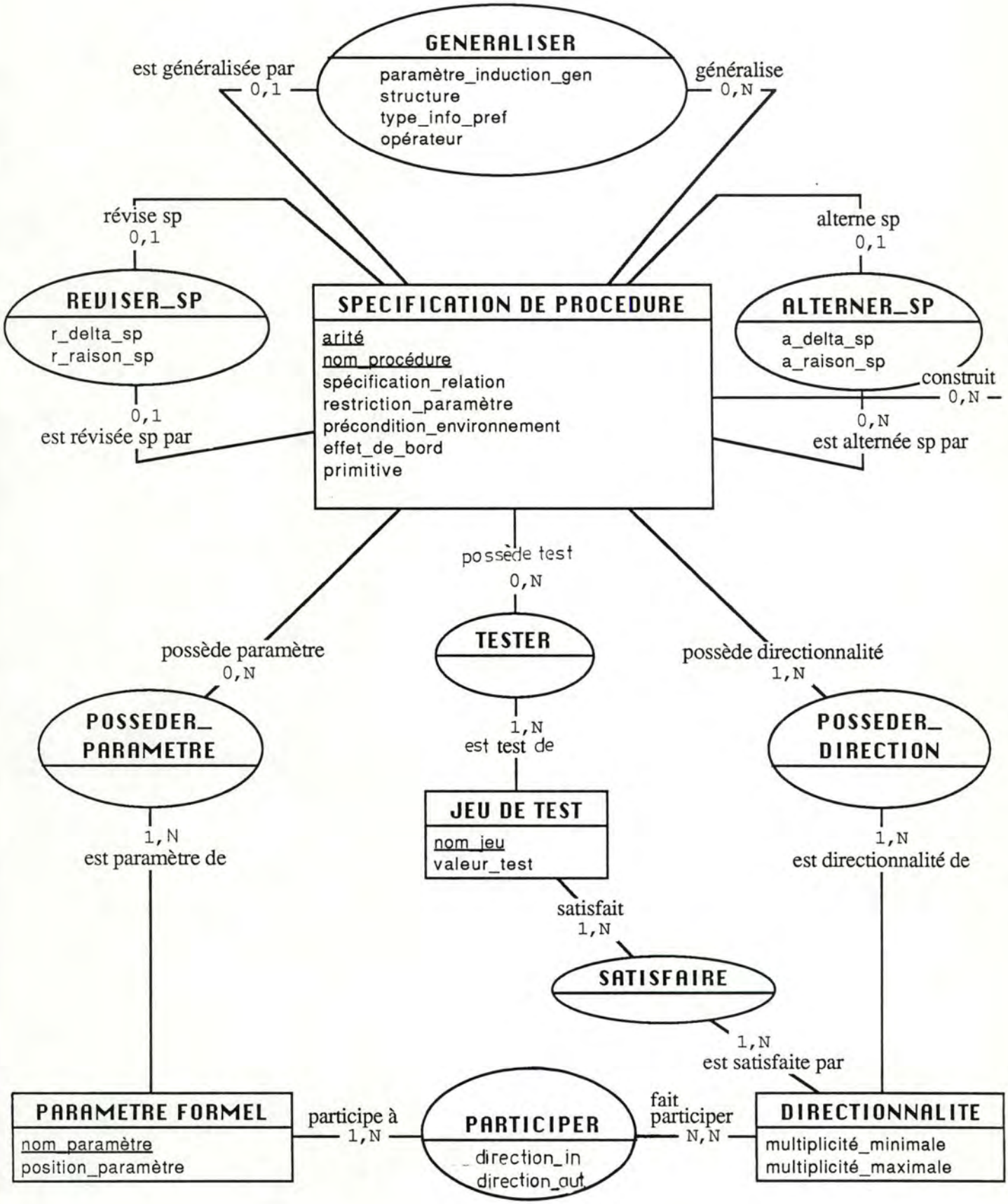
FIGURE 3.19 LES JEUX DE TEST.

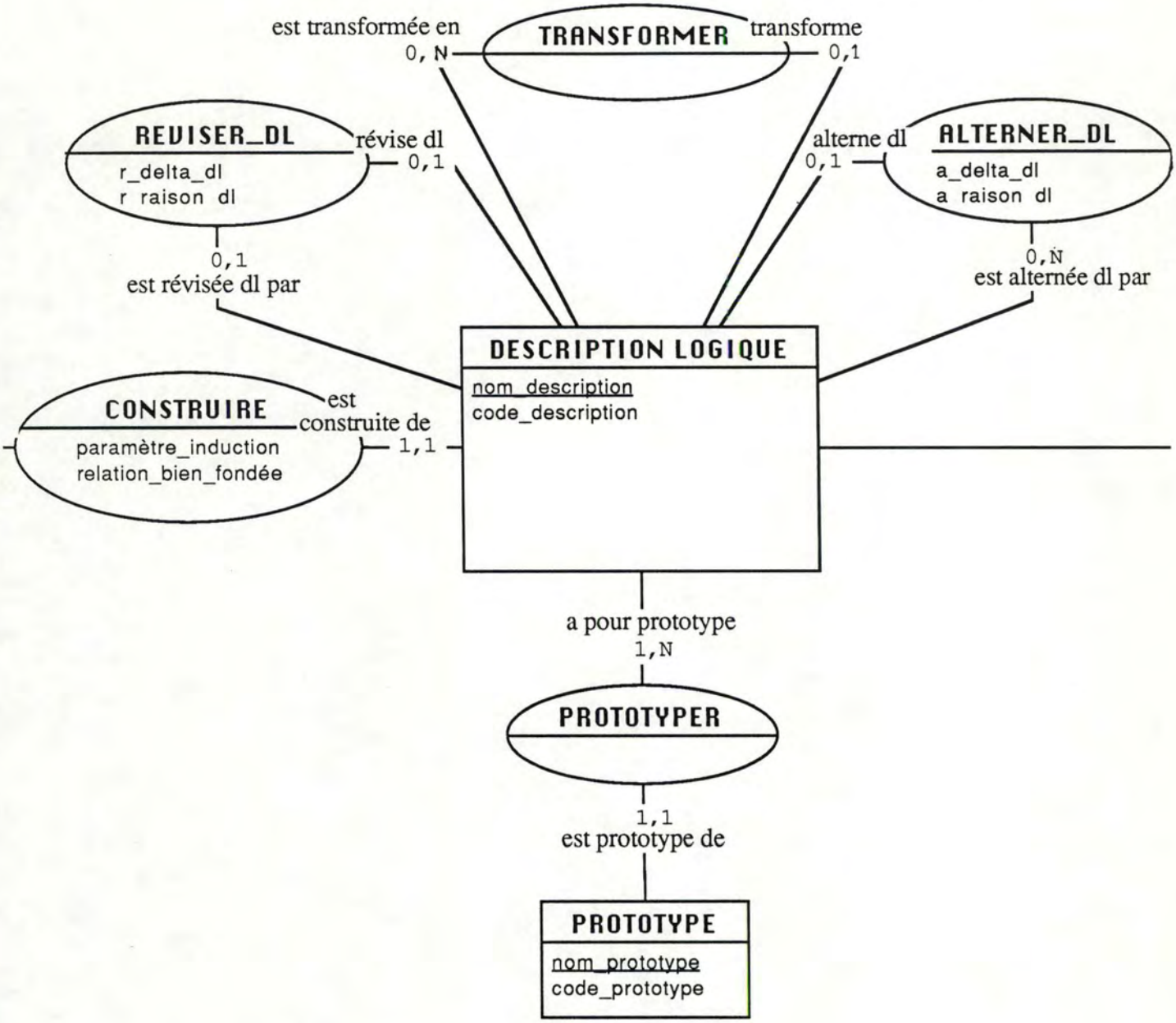
Contraintes :

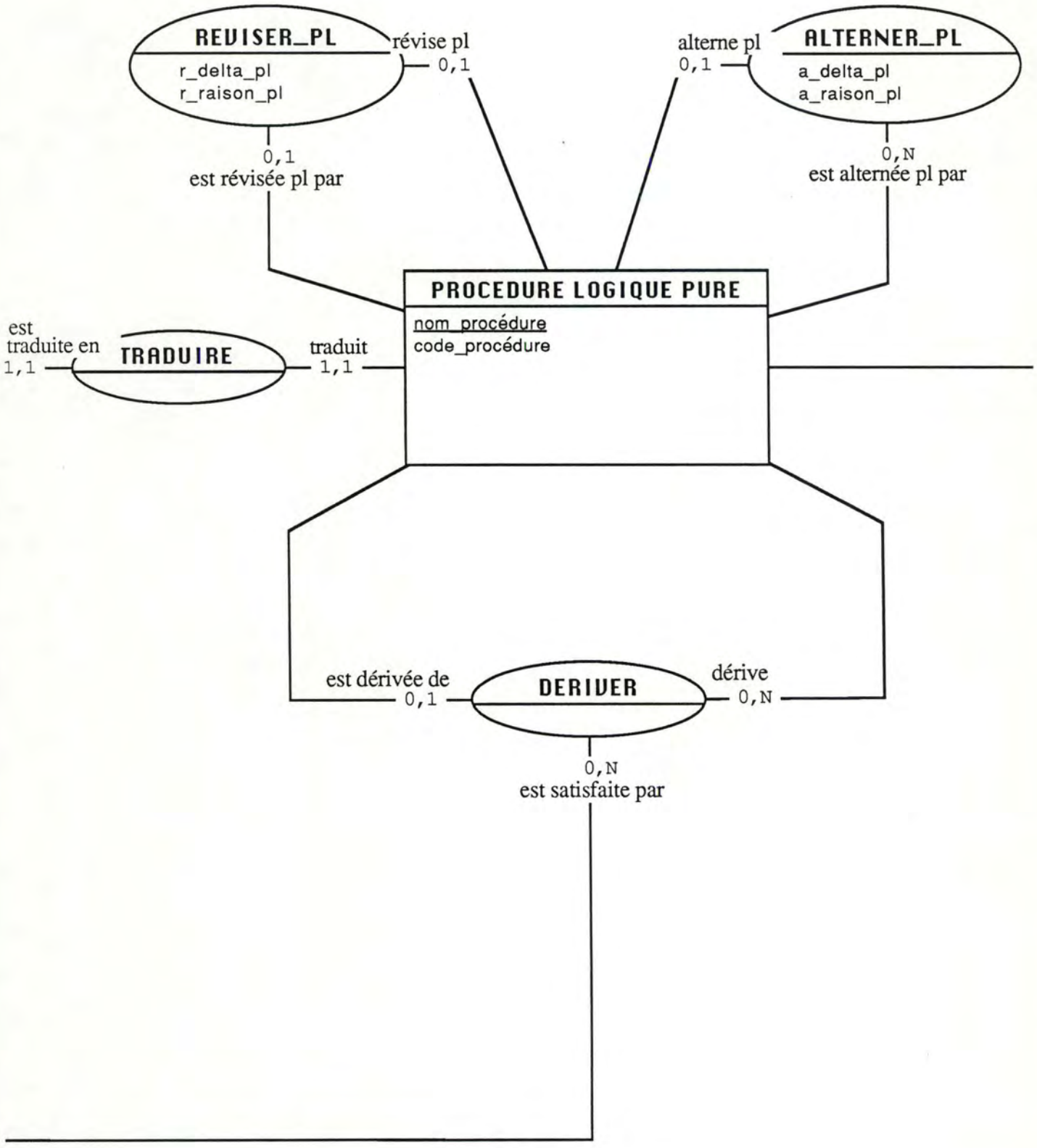
- la cardinalité maximale du rôle "satisfait" est plus petite ou égale au nombre de directionnalités de la spécification de procédure correspondante.

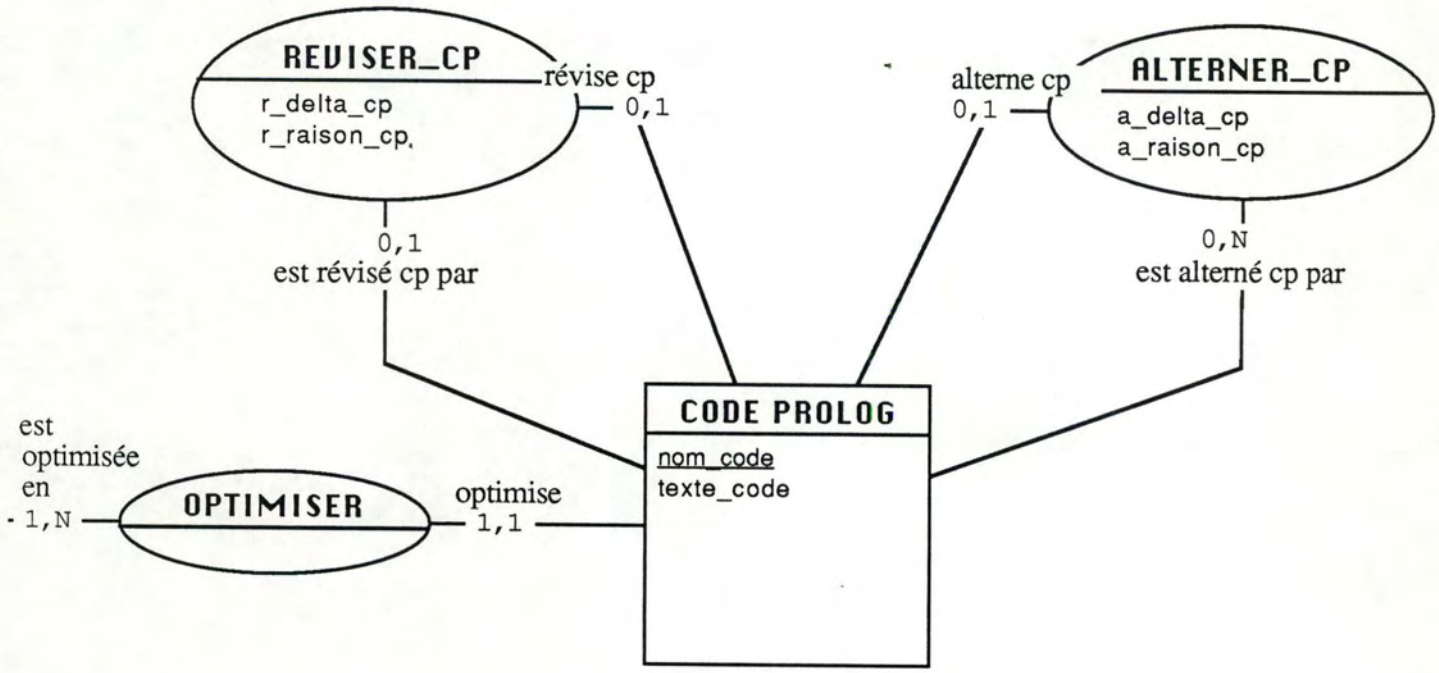
Le schéma complet du modèle E/R de la méthodologie est présenté dans les pages suivantes.


0,N
sous-type
SOUS-TYPER
0,N
est sous-typé par
RELATION BIEN FONDEE
om_relation
est définie par
1,N
DEFINIR
définit
1,N
TYPE
nom_type
description_type
type
0,N
TYPER
est typé par
1,1


GENERALISER
paramètre_induction_gen
structure
type_info_pref
opérateur
est généralisée par
0,1
généralise
0,N
révise sp
0,1
alterne sp
0,1
REVISER_SP
r_delta_sp
r_raison_sp
SPECIFICATION DE PROCEDURE
arité
nom_procédure
spécification_relation
restriction_paramètre
précondition_environnement
effet_de_bord
primitive
ALTERNER_SP
a_delta_sp
a_raison_sp
construit
0,N
0,1
est révisée sp par
0,N
est alternée sp par
possède test
0,N
possède paramètre
0,N
TESTER
possède directionnalité
1,N
POSSEDER_
PARAMETRE
1,N
est test de
POSSEDER_
DIRECTION
1,N
est paramètre de
JEU DE TEST
nom_jeu
valeur_test
1,N
est directionnalité de
satisfait
1,N
SATISFAIRE
1,N
est satisfaite par
PARAMETRE FORMEL
nom_paramètre
position_paramètre
participe à
1,N
PARTICIPER
direction_in
direction_out
fait
participer
N,N
DIRECTIONNALITE
multiplicité_minimale
multiplicité_maximale


est transformée en
0, N
TRANSFORMER
transforme
0,1
REVISER_DL
r_delta_dl
r raison dl
révise dl
0,1
alterne dl
0,1
ALTERNER_DL
a_delta_dl
a raison dl
0,1
est révisée dl par
0,N
est alternée dl par
DESCRIPTION LOGIQUE
nom_description
code_description
CONSTRUIRE
paramètre_induction
relation_bien_fondée
est
construite de
1,1
a pour prototype
1,N
PROTOTYPER
1,1
est prototype de
PROTOTYPE
nom_prototype
code_prototype


REVISER_PL
r_delta_pl
r_raison_pl
révise pl
0,1
0,1
est révisée pl par
alterne pl
0,1
ALTERNER_PL
a_delta_pl
a_raison_pl
0,N
est alternée pl par
PROCEDURE LOGIQUE PURE
nom_procédure
code_procédure
est
traduite en
1,1
TRADUIRE
traduit
1,1
est dérivée de
0,1
DERIVER
dérive
0,N
0,N
est satisfaite par


REVISER_CP
r_delta_cp
r_raison_cp
révise cp
0,1
0,1
est révisé cp par
alterne cp
0,1
ALTERNER_CP
a_delta_cp
a_raison_cp
0,N
est alterné cp par
CODE PROLOG
nom_code
texte_code
est
optimisée
en
1,N
OPTIMISER
optimise
1,1

3.3. Spécification d'un système de gestion de configurations.

Nous étudierons successivement dans cette section des procédures permettant l'identification des objets et leurs sélections et des procédures de contrôle des versions d'un objet. Cette étude permettra donc d'aborder une série de questions que pose tout système de gestion de configurations.

3.3.1. Le problème de l'identification des versions.

Notre volonté de gérer les versions multiples d'un objet nous impose de définir un système d'identification de ces versions. La majorité des environnements actuels utilisent un schéma d'identification des alternatives et des révisions décrit par un triplet :

(nom_de_l'_objet, alternative, révision)

où :
- . nom_de_l'_objet correspond au nom usuel d'un objet,
- . alternative sélectionne un membre de l'ensemble des objets,
- . révision sélectionne une des révisions d'une alternative.

Nous allons présenter maintenant quelques techniques d'identification les plus couramment utilisées :

- Les versions sont identifiées par un nom et une estampille correspondant à la date de création de l'objet. Il n'y a pas d'autres caractéristiques pour décrire les alternatives et les révisions. De ce fait, cette technique qui ne permet d'ailleurs pas de distinguer révision et alternative nous semble quelque peu pauvre d'utilisation !

- On peut désigner l'alternative par un identificateur et les révisions par un nombre attribué séquentiellement par le système de gestion de version.

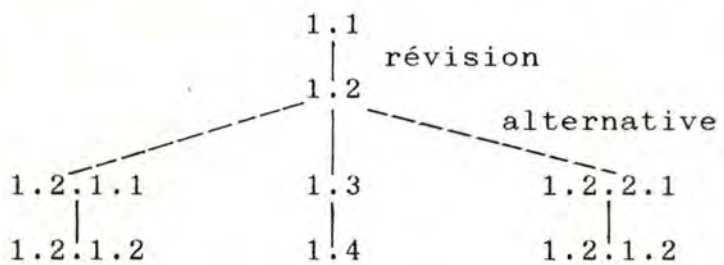
Exemple 3.1 : Désignation par identification et nombre.

```
Alternative1 : 01 ----> 02 -----> 03 ----> 04
Alternative2 : 01 ----> 02 ----> 03
Alternative3 : 01 ----> 02 ----> 03
```

■

- On peut, enfin, désigner les alternatives et les révisions par une numérotation hiérarchique. Les révisions sont organisées en arbre où les branches sont les alternatives. Une révision est identifiée par deux nombres : le "release number" et le "level number". Une alternative est identifiée par le numéro de la révision commune, un chiffre pour l'alternative et un chiffre pour la révision de cette alternative. Par révision commune, nous entendons la révision qui donne lieu à une alternative.

Exemple 3.2 : Désignation par une numérotation hiérarchique.



■

Ce mode de désignation, permettant de se retrouver rapidement dans l'arbre des versions peut s'avérer très complexe lorsqu'un objet tend à évoluer en de nombreuses alternatives (on peut ainsi atteindre 8 à 10 chiffres !)

Notre système d'identification des objets est défini suivant la syntaxe :

<nom_objet> :: = <objet> / <type> [: <alternative>. <révision>]

<type> :: = sp | dl | pl | cp

<révision> :: = 1...999

où <objet> et <alternative> sont des identificateurs, cette notion va être précisée dans la suite.

type = sp désigne un objet spécification de procédure,

type = dl désigne un objet procédure logique,

type = pl désigne un objet procédure logique pure,

type = cp désigne un objet code prolog.

Exemple 3.3 Désignation choisie.

compress / dl : desc. 015 désigne la révision 15 de l'alternative desc de l'objet description logique compress.

■

Nous allons gérer les alternatives sous forme de suites parallèles de révisions.

La numérotation séquentielle des révisions est maintenue automatiquement par le système de gestion de configurations. La révision originale possède le numéro 1 et les révisions successives sont numérotées 2,3... etc. La désignation d'un objet ou d'une alternative peut être soit laissée sous la responsabilité de l'utilisateur, soit réalisée automatiquement par le système.

Nous allons détailler notre stratégie de désignation.

Dans le cas d'une désignation manuelle, un identificateur d'objet ainsi qu'un indentificateur d'alternative doit être une suite de caractères parmi {A..Z, a..z, 0..9, _} . Dans le cas de l'identification automatique d'un objet, le système lui attribuera la suite de caractères "OBJ_" suivie d'un numéro comptabilisant le nombre d'objets déjà créés automatiquement.
De la même façon, une alternative d'un objet sera identifié par la suite de caractères "ALT_ suivie d'un numéro comptabilisant le nombre d'alternatives créés automatiquement.
Nous permettons en outre l'utilisation de deux méta-caractères, notés "*" et "?" dans les identificateurs d'objets et d'alternatives. Le méta-caractère "*" correspond à une suite quelconque de carctères parmi {A.. Z, a..z, 0..9, _} tandis que le méta-caractère "?" correspond à un caractère parmi cet ensemble.

Exemple 3.4 Désignation d'un objet avec les méta-caractères.

compress */dl : desc . 015 désigne la révision 15 de l'alternative desc des objets descriptions logiques ayant compress comme préfixe et un suffixe formé d'un nombre quelconque de caractère.

compress ?/dl : desc . 015 désigne la révision 15 de l'alternative desc des objets descriptions logiques ayant compress comme préfixe et un suffixe formé d'un carctère.

■

En ce qui concerne le type d'objet, il peut aussi être généré manuellement ou automatiquement lors de la conservation de l'objet. Dans le premier cas, une vérification de cohérence doit être réalisée entre le type et le lieu de conservation dans la base. Dans le second cas, il est ajouté par le système selon l'endroit de la base d'objet où l'on

veut conserver cet objet.

3.3.2. Mécanisme de sélection de versions.

Après avoir précisé notre méthode d'identification des versions, nous voudrions décrire un mécanisme de sélection des versions. Nous allons identifier trois manières de sélectionner une version :

- sélection en extension,
- sélection par défaut
- sélection en intention.

Nous remarquerons que, quelque soit le mécanisme de sélection choisi, l'utilisation de méta-caractères dans les identifiants permet de sélectionner un ensemble d'objets.

3.3.2.1 Sélection en extension.

C'est la solution la plus évidente. Elle consiste à énumérer complètement la version de l'objet que l'on veut obtenir.

Exemple 3.5 Sélection en extension.

sélection de compress / dl : desc . 15
revient à sélectionner la révision 15 de l'alternative desc de l'objet description logique.

■

3.3.2.2. Sélection par défaut.

Pour un objet, la règle de sélection par défaut consiste à sélectionner la dernière révision de la première alternative de cet objet. Pour les alternatives, le choix par défaut de la première, qui correspond dans la désignation hiérarchique à l'évolution linéaire, nous semblait la plus pertinente.

Dans le cas où le type de l'objet n'est pas spécifié, la sélection se fait d'abord parmi les spécifications de procédure. Si elle ne réussit pas, la sélection se fera parmi les descriptions logiques et ainsi de suite pour les procédures logiques pures et les codes PROLOG.

Règles de sélection par défaut

1. sélection de objet/type_objet correspond à sélection objet/type_objet : alt1. dernière_rev.
2. sélection de objet/type_objet : alt correspond à sélection de objet/type_objet : alt . dernière_rev
3. sélection de objet/type_objet : rev correspond à sélection de objet/type_objet : alt1. rev
4. sélection de objet : alt . rev correspond à sélection de objet/type_objet : alt.rev

   où type_objet prend successivement tant qu'une sélection ne réussit pas les valeurs sp, dl, pl, cp
5. sélection de objet : alt correspond à sélection de objet/type_objet : alt.dernière_rev

   où type_objet prend succesivement tant qu'une sélection ne ne réussit pas les valeurs sp, dl, pl, cp
6. sélection de objet. rev correspond à sélection de objet/type_objet: alt1. rev

   où type_objet prend successivement tant qu'une sélection ne réussit pas les valeurs sp, dl, pl, cp

Défintion en intention.

Rappelons tout d'abord que chaque objet est caractérisé par un ensemble d'attributs, c'est-à-dire un ensemble de propriétés qualifiant cet objet. Un attribut est défini par un nom et un ensemble de valeurs. Une version d'un objet peut être sélectionnée selon les conditions sur les valeurs de ces attributs par un langage comparable à un langage de requête à une base de données. Pour cela, il faut que les attributs aient des valeurs bien déterminées, ce qui n'est pas le cas dans notre modèle. Nous laisserons donc cette sélection pour un développement ultérieur.

3.3.3. Le contrôle des versions.

La fonction de contrôle de versions crée pour chaque modification une nouvelle version d'un objet. Les services identifiés et automatisés pour le contrôle de versions sont : l'optimisation du stockage du contenu des versions, l'enregistrement de l'historique d'évolution, le contrôle de l'accès aux versions et la fusion des versions. Nous allons faire le point sur l'état de développement de chacun des services.

3.3.3.1. La gestion de l'espace d'archivage : le mécanisme des deltas.

Un des nombreux problèmes que nous pose la gestion de versions est celui de la conservation des versions (révisions et alternatives) d'un objet sous la forme d'un texte source. Une première solution est d'enregistrer chaque version de cet objet.

Cette solution, d'une simplicité de réalisation, possède

l'énorme désavantage de consommer beaucoup d'espace de stockage !

Nous nous tournerons vers une seconde solution de stockage, le mécanisme de delta (Rochkind 75), (Tichy 85). Son principe est d'enregistrer une seule fois une version complète d'un objet et de représenter les autres versions par leurs différences à partir de cette version complète. Le mécanisme de delta est généralement basé sur la ligne, cela signifie que les différences entre deux versions correspondent à l'insertion ou à la suppression de plusieurs lignes.

Exemple 3.6 : Delta entre deux textes source.

```
Texte A = 1  while not eof do      Texte B = 1  while not eof do
          2                                  2     begin
          3        read(f,car);              3        read(f,car);
          4        write(car);               4        writeln(car)
          5     end;                         5     end;
```

le delta du texte du texte_A au texte_B : supprimer la ligne 2

insérer à la ligne 2, la ligne : begin

supprimer la ligne 4

insérer à la ligne 4, la ligne : writeln(car)

■

L'application du mécanisme du delta conduit à un conflit entre l'espace de stockage et le temps d'accès. En effet, s'il est prouvé que les deltas réduisent considérablement l'espace de conservation des multiples versions d'un objet, ils peuvent aussi augmenter le temps d'accès aux versions. Car pour obtenir une version, il faut reprendre la première version complète et lui appliquer les deltas successivement.
Cependant, avec les méthodes actuelles (Tichy 85), le temps

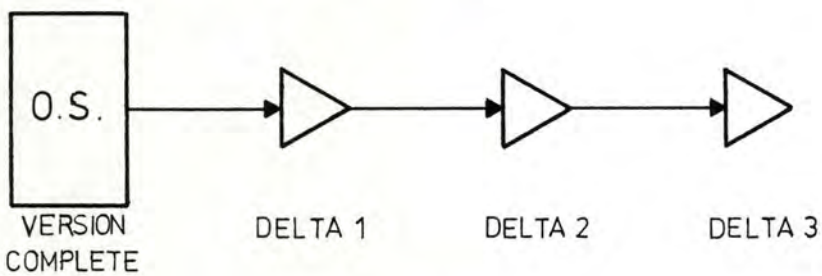
d'accès à une version est de l'ordre du temps d'accès direct à la version.

Nous allons mettre en évidence dans la suite différentes techniques d'application des deltas.

- les deltas avant.

Avec cette technique, nous disposons de la plus ancienne version dans sa forme complète. Les deltas représentent les transformations de la version I à la version I + 1. En fonction des fréquences d'accès aux dernières versions, cette technique peut s'avérer peu efficace car il faut appliquer une série souvent longue de deltas pour obtenir ces dernières versions.

Exemple 3.7 Deltas avant.



Un objet O.S. a été modifié trois fois mais c'est l'original qui est conservé intégralement.

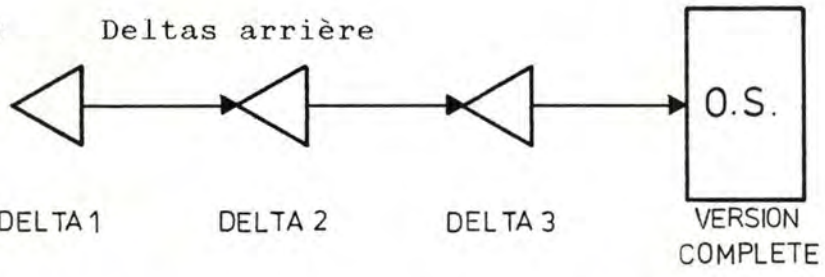
■

- les deltas arrière.

Disposant de la version la plus récente, les deltas représentent les transformations de la version I à la version I - 1. Si l'on accède souvent aux dernières versions, cette technique optimise le temps d'accès. Elle correspond en effet

à une simple opération de copie. L'ajout d'une nouvelle version ne demande le calcul que d'un seul delta.

Exemple 3.8 Deltas arrière



Un objet O.S. a été modifié trois fois mais c'est le dernier qui est conservé intégralement.

■

- les deltas interfoliés.

Chaque objet est enregistré dans un seul fichier séquentiel. Ce fichier est constitué d'une part d'enregistrements de texte, un par ligne, contenant le code source inséré par les deltas et d'autre part d'enregistrements de contrôle qui spécifient les effets de chaque delta. Il y a ainsi trois types d'enregistrement de contrôle : I(nsertion), E(ffacement) et F(in) suivis d'un identificateur de version. Une version particulière est générée de la façon suivante : le fichier est parcouru séquentiellement, enregistrement par enregistrement; lorsqu'on rencontre du texte, il est soit retenu, soit sauté selon que l'enregistrement de contrôle satisfait à la version en cours. N'importe quelle version d'un texte est ainsi reconstruite en une passe. Le temps de reconstruction est identique quelque soit la version demandée.

Exemple 3.9 Deltas interfoliés.

I.1.1
I.1.4
a_texte 1.4
b_texte 1.1
E 1.2
c_texte 1.1
F 1.2
d_texte 1.2
E 1.3
e_texte 1.2
F 1.2
f_texte 1.1
F 1.3
g_texte 1.1
F.1.1

version 1.1	version 1.2	version 1.3	version 1.4
b_texte 1.1	b_texte 1.1	b_texte 1.1	a_texte 1.4
c_texte 1.2	d_texte 1.2	d_texte 1.2	b_texte 1.1
f_texte 1.1	e_texte 1.2	g_texte 1.1	d_texte 1.2
g_texte 1.1	f_texte 1.1		g_texte 1.1
	g_texte 1.1		

■

Nous choisirons, en raison de sa rapidité d'accès aux dernières versions une technique de gestion des deltas semblable à celle du Revision Control System (RCS) (Tichy 85).

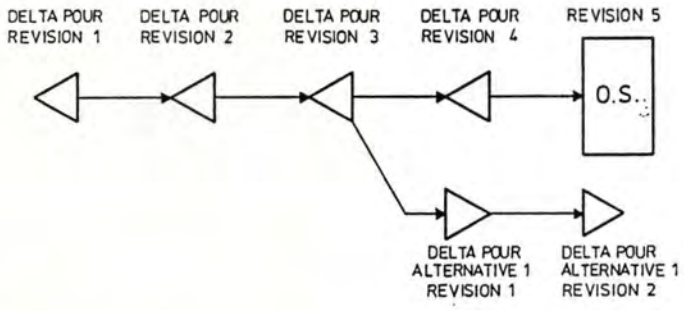
L'utilisation de RCS sur une période de 13 mois a montré, qu'en moyenne, 1,02 deltas sont appliqués par accès à une version. Les opérations demandant le calcul d'un delta ne représentaient que 5% des opérations globales. Cela veut donc dire que dans 95% des cas, un accès à la dernière révision est demandé.

Les alternatives demandent une réflexion un peu différente. La première idée voudrait que l'on conserve aussi intacte la dernière alternative.

Cette approche gaspille l'espace de stockage. En effet, l est clair qu'une alternative peut être calculée à partir de sa révision commune. Cette révision est donc considérée comme la version complète de l'alternative et la technique des deltas avant peut être appliquée sur les alternatives.

Enfin, pour obtenir un gain d'espace de stockage supplémentaire, nous combinerons la gestion des deltas avec une technique de comparaison des blancs.

Exemple 3.10 Gestion des versions par deltas RCS.



■

Les mesures effectuées montrent qu'un fichier de deltas dont la taille est celle du code source peut contenir en moyenne de 50 à 100 (parfois 200) révisions du code source (Leblang 88).

3.3.3.2. **La gestion d'historique.**

Nous voudrions, pour tout objet évoluant en versions, tenir à jour un historique d'évolution. Il peut être créé et maintenu automatiquement par l'environnement lors de l'enregistrement de la nouvelle version.

Un historique conserve une trace des transformations successives effectuées sur un objet. Il est constitué d'un ensemble d'informations précises et pertinentes comme par exemple :

- le nom de la personne responsable des modifications,
- la date et l'heure de l'enregistrement des modifications,
- l'objet qui a subi les modifications,
- un commentaire expliquant ce qui a été modifié, pourquoi et comment les modifications ont été réalisées.

Nous avons prévu d'extraire pour un objet, soit son historique complet, soit l'historique de la dernière modification. Une autre fonction doit permettre d'incorporer le dernier historique à une version d'un objet, sous forme de commentaires.

3.3.3.3 **Les modifications concurrentes d'un objet.**

Un des premiers principes de la gestion de configurations est d'éviter les copies multiples d'un même objet. En effet, dans le monde réel d'un projet de programmation en équipe, rien ne peut garantir que deux copies identiques le resteront toujours. Les copies multiples divergent inévitablement.

Cette situation inacceptable est aisément évitée si l'on conserve en un seul endroit la base d'objets. Les analystes peuvent alors faire une copie temporaire d'un

objet dans un espace privé afin de pouvoir appliquer les outils classiques de traitement. Généralement, les analystes communiqueront avec la base d'objets en déplaçant les données depuis ou vers leur espace de travail.

Un problème survient dès lors que deux développeurs effectuent des mises-à-jour sur même objet dans leur espace de travail, une des deux modifications sera inévitablement perdue si des précautions ne sont pas prises. Ce problème, devenu classique, a été posé et bien résolu dans RCS (Tichy 85).

La solution consiste à ne permettre qu'à un analyste de copier un objet pour une mise-à-jour, tout en permettant aux autres développeurs de consulter cet objet. En copiant un objet de la base pour une mise-à-jour, l'analyste place un verrou en écriture sur cet objet par une commande de réservation. Lorsqu'il recopie l'objet modifié de son espace de travail dans la base, une nouvelle version est créée dans la base et le verrou est alors libéré.

Deux commandes, "réserver" et "libérer", permettent le développement sans interférence d'actions concurrentes de manipulation de la base. Un objet ne peut être modifié que par l'usager qui l'a réservé.

3.3.3.4 Le mécanisme de fusion.

Lorsqu'un objet est déjà réservé et qu'un autre membre du projet tente de le réserver, notre système le prévient. Cependant, il peut y avoir des raisons à ce que plusieurs personnes travaillent concurremment sur le même élément. Par exemple pour des contraintes des temps, les changements sont opérés en parallèle alors que logiquement, ils se suivent.

Une fois que des alternatives existent, nous voudrions qu'il soit possible de fusionner ces variantes en une seule branche. Ainsi, le travail fait séparément peut être mis en commun. L'automatisation de la fusion des changements effectués en parallèle n'a pas de solution générale. Il n'est possible d'assister la fusion que dans des cas simples, par exemple la conformité par rapport à la syntaxe.

3.2. Evaluation du travail

La majeure partie de ce travail a consisté en une présentation incrémentale d'un modèle Entité/Relation de la méthodologie de développement en programmation logique. Nous nous sommes efforcés de réflèter le plus fidèlement possible dans cette modélisation les principaux concepts présentés et explicités dans cette méthodologie.

Des extensions possibles pourraient entre autre s'intéresser à

- exprimer dans le schéma les relations qui peuvent exister entre certains paramètres,
- exprimer les formes structurelles génériques définies par chaque type de paramètres.

Les choix, effectués dans la spécification de notre système de gestion de configurations, ont essayé de tenir compte des solutions les plus performantes parmi les tendances actuelles en la matière. Cependant, il nous semble que d'autres solutions pourraient être envisagées avec autant de réussite.

Nous sommes conscients de n'avoir pas spécifié toutes les fonctionnalités d'un gestionnaire de configuration. Sont laissés, par exemple, à des développements ultérieurs :

- l'étude approfondie des mécanismes de sélection de versions
- l'étude de technique de fusion de versions,
- l'étude d'un gestionnaire des dépendances permettant de déterminer les objectifs atteints par la propagation des

changements et les actions à réaliser sur ces objets. Plus particulièrement, la définition d'un langage de description des dépendances nous semble importante à envisager.

En guise de conclusion, nous décrivons les aspects abordés dans cette étude puis nous situons notre contribution au travail réalisé. Enfin, nous présentons les perspectives et les nouveaux thèmes de développement et de recherche possibles.

1. Aspects abordés.

Nous avons présenté un modèle simplifié de cycle de vie en programmation impérative ainsi qu'un outil d'aide à la programmation.

Fort de cette expérience, nous avons ensuite présenté un modèle de cycle de vie en programmation logique et les principales fonctionnalités d'un système de gestion de configurations.

Les aspects étudiés dans ces travaux sont liés à l'identification des objets logiciels, à la représentation de cet ensemble d'objets dépendants les uns des autres et à la définition de mécanismes permettant de contrôler l'évolution en versions multiples de ces objets.

2. Contributions personnelles.

Le modèle simplifié du cycle de vie et l'outil d'aide à la programmation résulte de mon travail de stage à Lausanne.

La méthodologie de développement de programme logique sur laquelle se base ma modélisation Entité/Relation est

présentée dans la thèse de doctorat de Yves Deville (Deville 89).

Les choix qui m'ont conduit à spécifier un système de gestion de configurations proviennent de nombreuses sources (Belkhatir 88), (Leblang 88), (Rockhind 75), (Tichy 85).

3. Perspectives

A partir de cette étude, des activités de développement et de recherche peuvent être envisagées.

Une couverture plus générale du cycle de vie en programmation impérative devrait être prise en compte dans notre modèle. Des améliorations de l'outil proposé peuvent inclure :

- son indépendance vis-à-vis d'un langage de programmation particulier,
- son intégration dans un atelier logiciel.

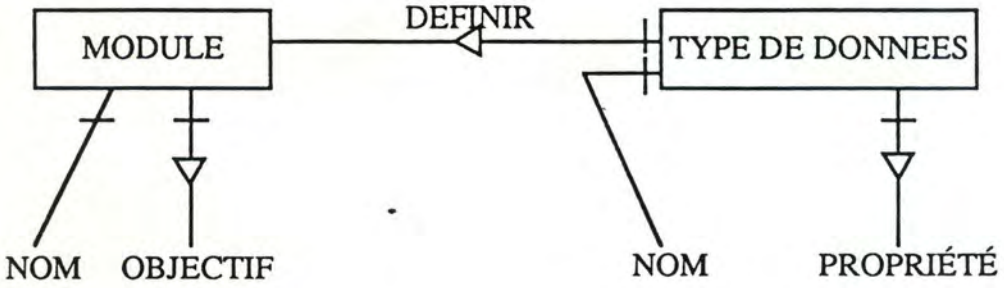
Des extensions au modèle de cycle de vie en programmation logique pourraient s'intéresser à l'introduction explicite de relations entre certains objets logiciels.

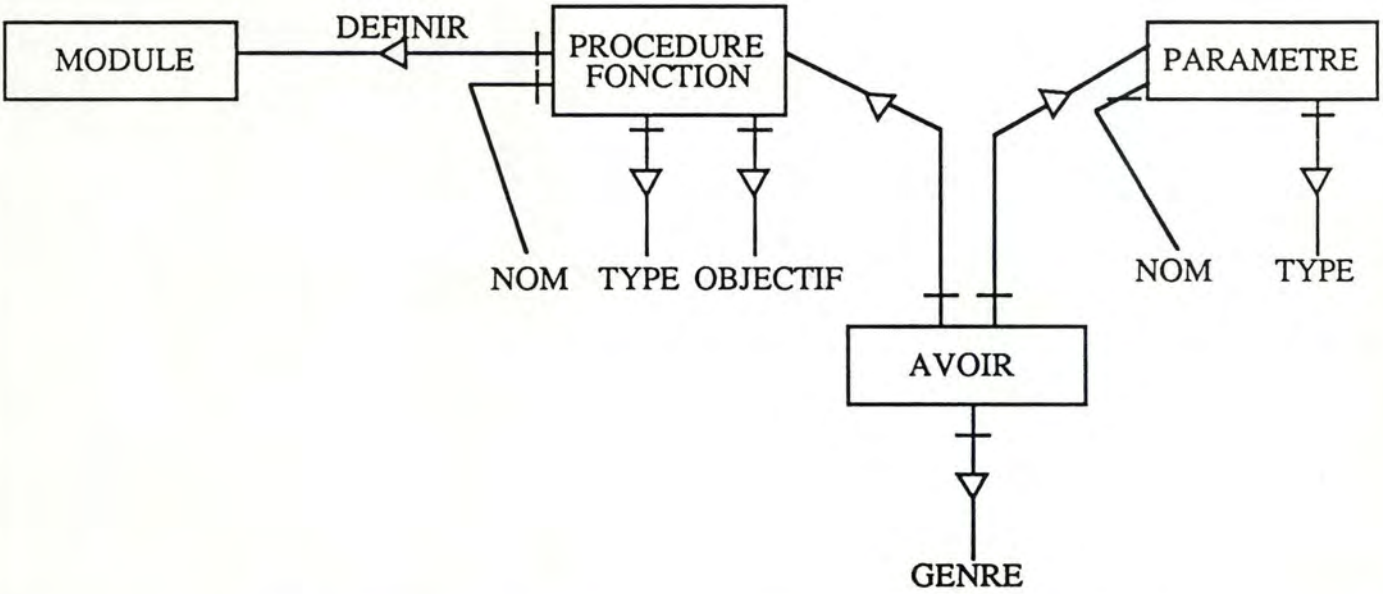
Enfin, des améliorations et extensions sont à apporter au système de gestion de configurations sur :

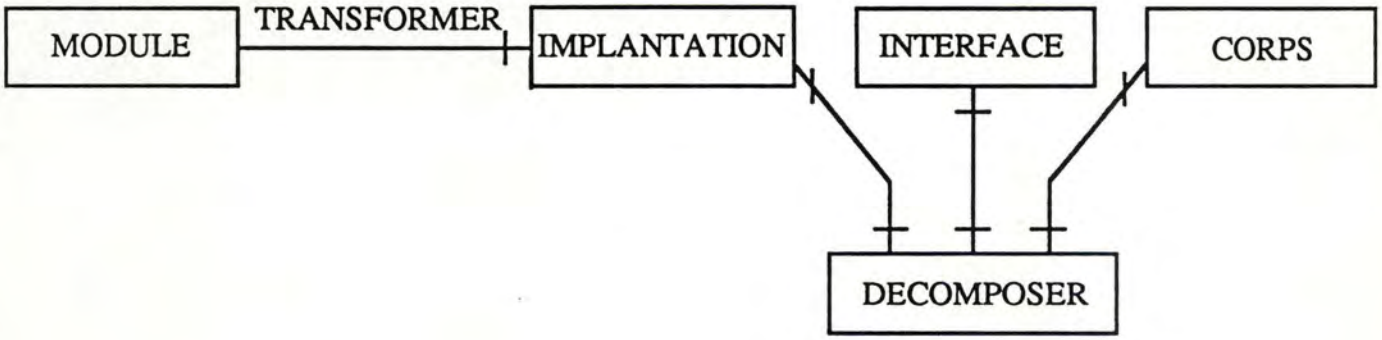
- les mécanismes de sélection d'objets,
- la fusion de version d'objets,
- les mécanismes nécessaires pour apprécier et gérer les impacts d'une modification d'un objet.

ANNEXE 1 :

SCHEMA DU MODELE D'ACCES GENERALISE .


MODULE
DEFINIR
TYPE DE DONNEES
NOM
OBJECTIF
NOM
PROPRIÉTÉ

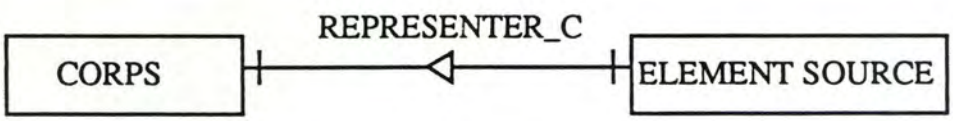

MODULE
DEFINIR
PROCEDURE
FONCTION
PARAMETRE
NOM
TYPE
OBJECTIF
NOM
TYPE
AVOIR
GENRE

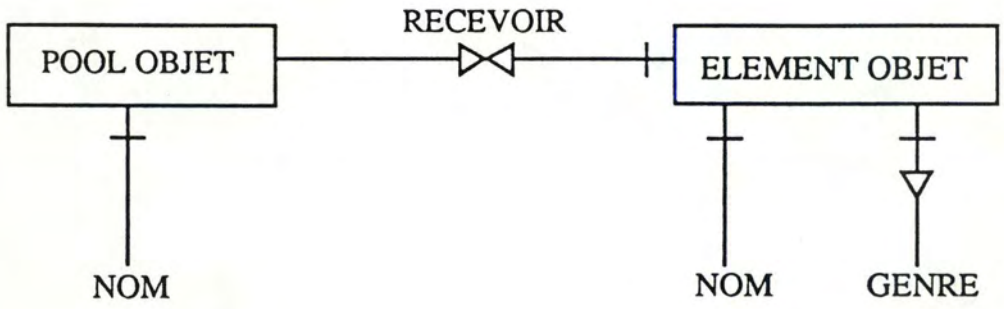

MODULE
TRANSFORMER
IMPLANTATION
INTERFACE
CORPS
DECOMPOSER

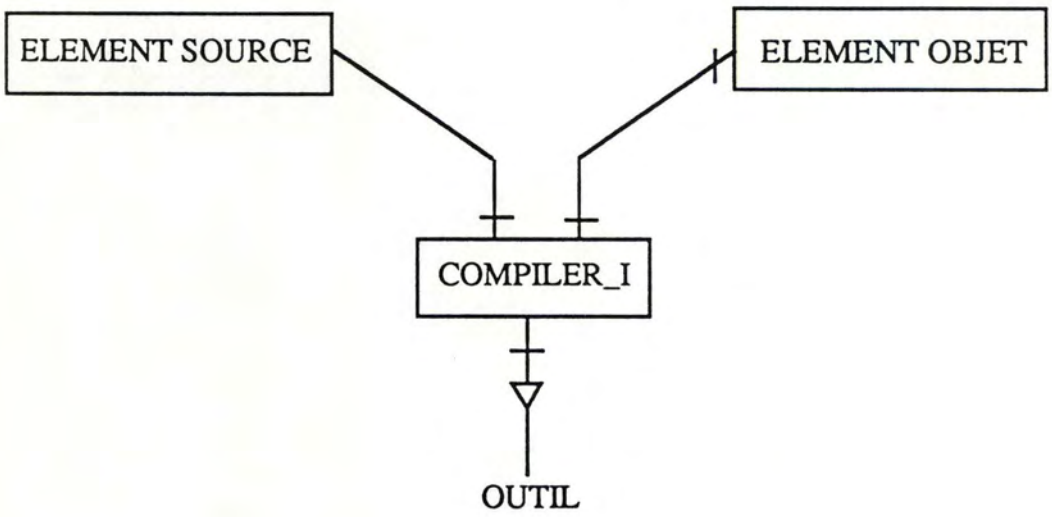
MODULE
UTILISER
TYPE DE DONNEES

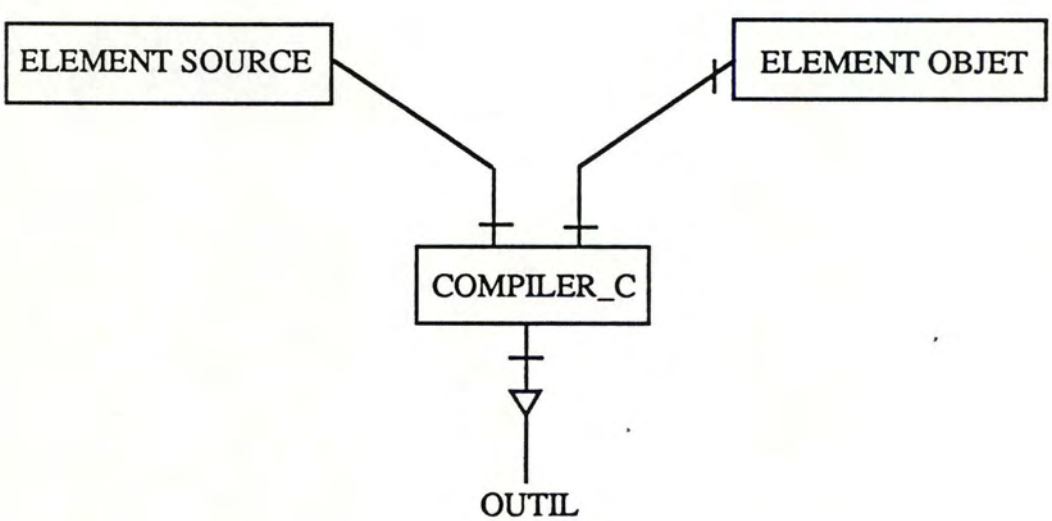
MODULE
UTILISER
PROCEDURE
FONCTION

INTERFACE
REPRESENTER_I
ELEMENT SOURCE

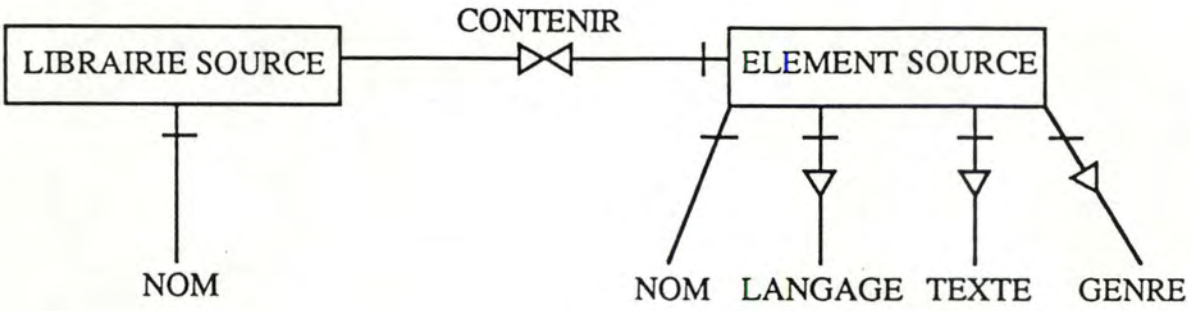

CORPS
REPRESENTER_C
ELEMENT SOURCE

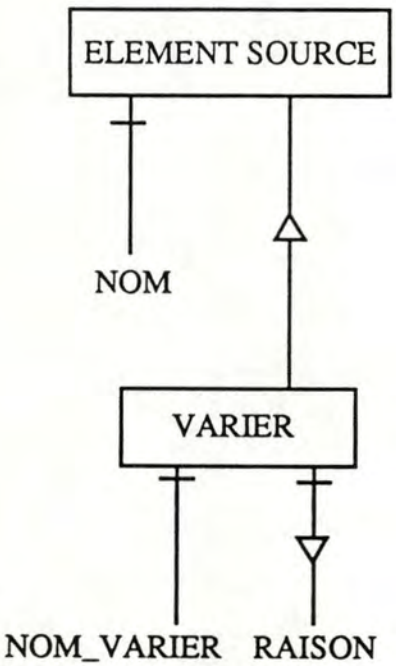

POOL OBJET
RECEVOIR
ELEMENT OBJET
NOM
NOM
GENRE

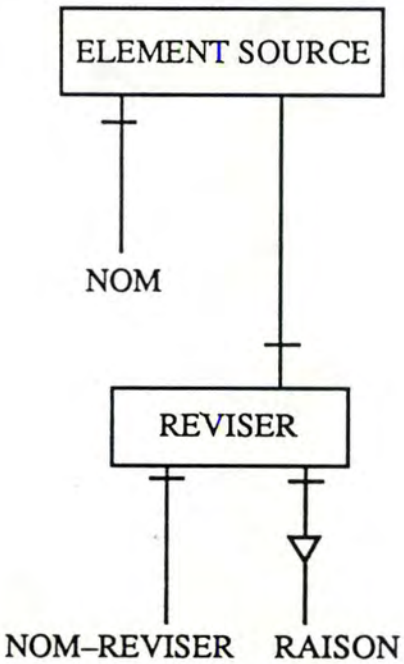

ELEMENT SOURCE
ELEMENT OBJET
COMPILER_I
OUTIL

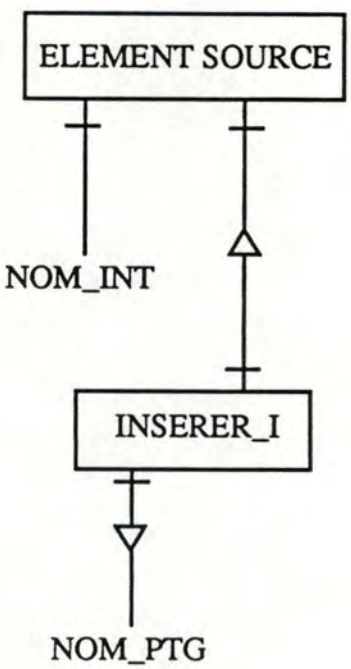

ELEMENT SOURCE
ELEMENT OBJET
COMPILER_C
OUTIL

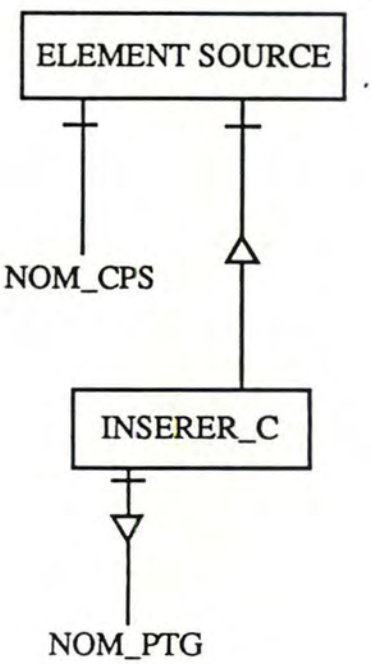
ELEMENT SOURCE
HERITER
ELEMENT OBJET


LIBRAIRIE SOURCE
CONTENIR
ELEMENT SOURCE
NOM
NOM
LANGAGE
TEXTE
GENRE


ELEMENT SOURCE
NOM
VARIER
NOM_VARIER
RAISON

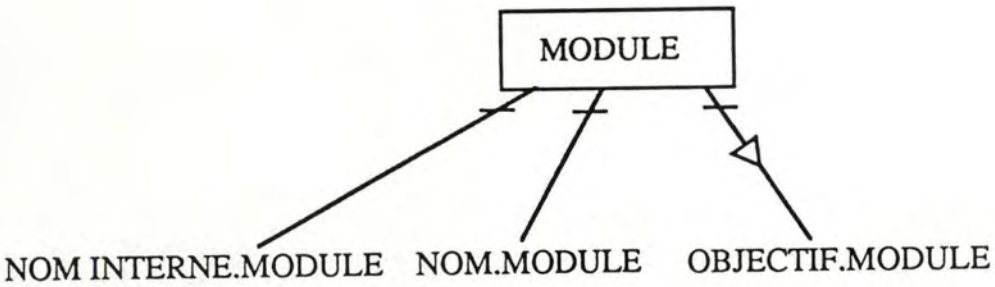

ELEMENT SOURCE
NOM
REVISER
NOM–REVISER
RAISON

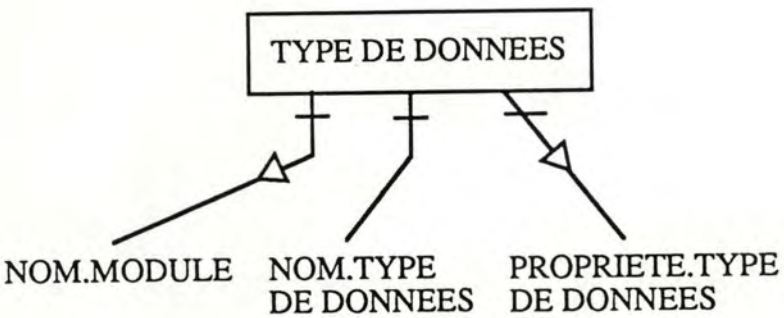

ELEMENT SOURCE
NOM_INT
INSERER_I
NOM_PTG

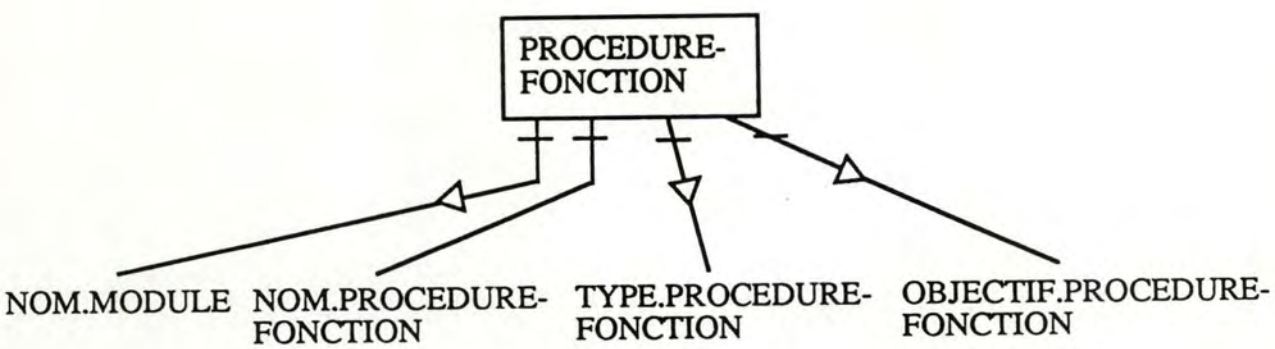

ELEMENT SOURCE
NOM_CPS
INSERER_C
NOM_PTG

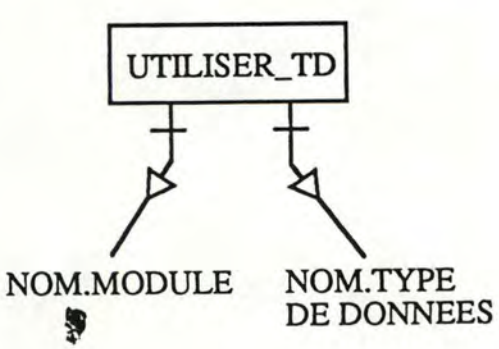
ANNEXE 2 :

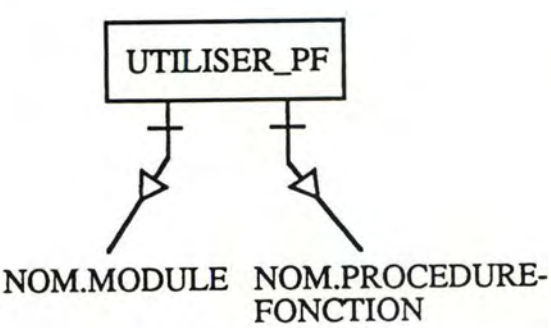
SCHEMA CONFORME AU MODELE RELATIONNEL.

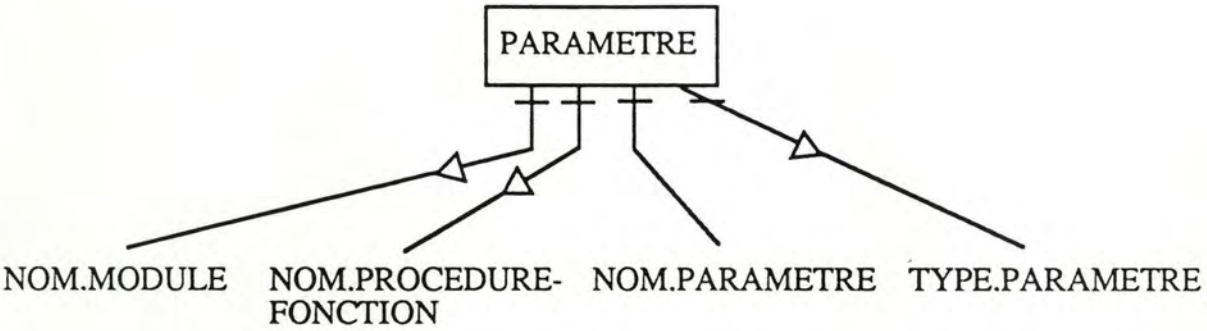

MODULE
NOM INTERNE.MODULE
NOM.MODULE
OBJECTIF.MODULE

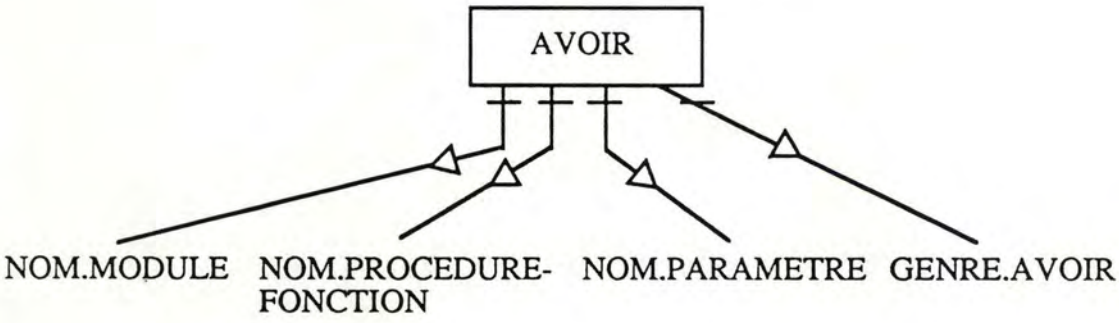

TYPE DE DONNEES
NOM.MODULE
NOM.TYPE DE DONNEES
PROPRIETE.TYPE DE DONNEES

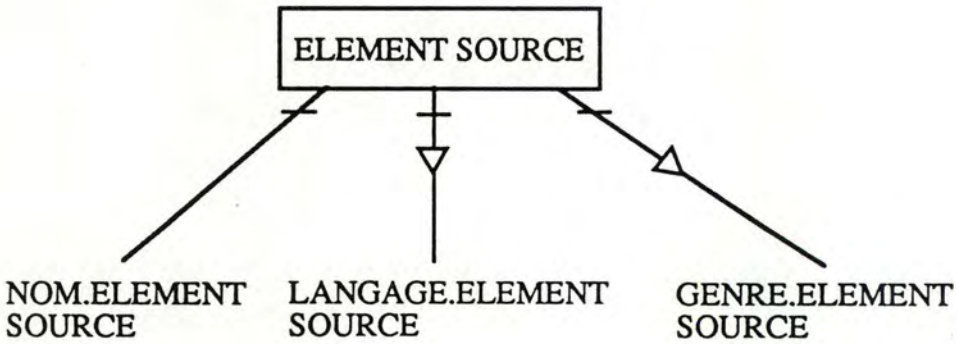

PROCEDURE-FONCTION
NOM.MODULE
NOM.PROCEDURE-FONCTION
TYPE.PROCEDURE-FONCTION
OBJECTIF.PROCEDURE-FONCTION

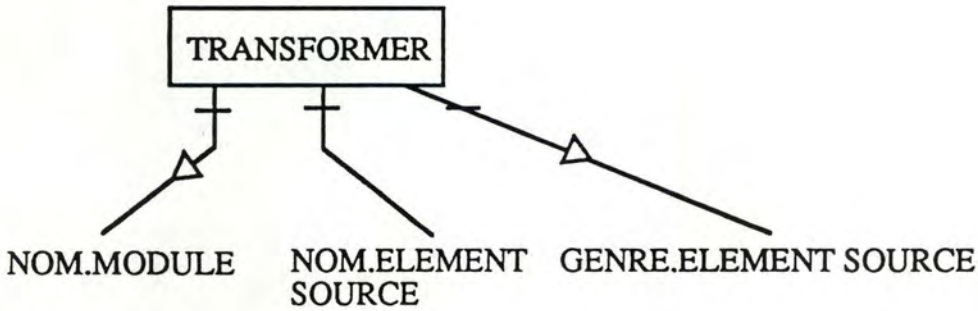

UTILISER_TD
NOM.MODULE
NOM.TYPE DE DONNEES

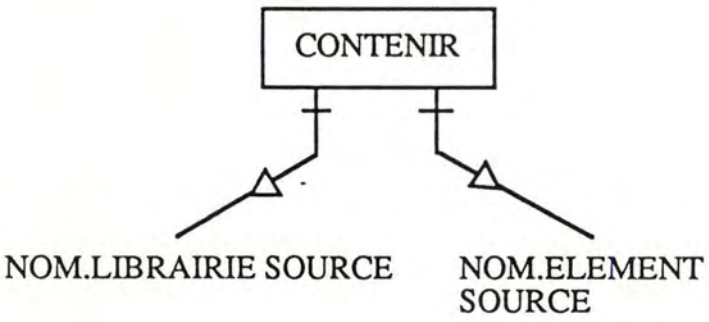

UTILISER_PF
NOM.MODULE
NOM.PROCEDURE-FONCTION

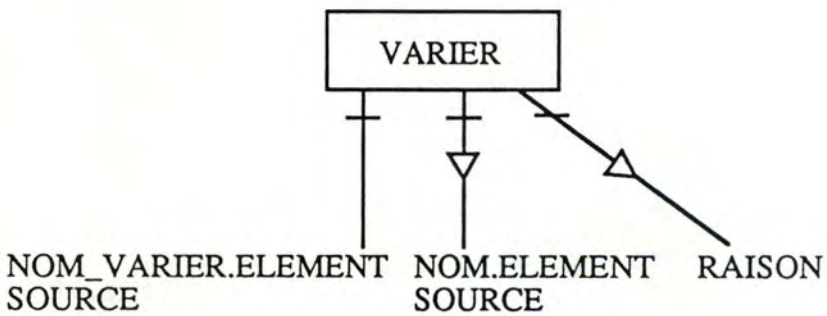

PARAMETRE
NOM.MODULE
NOM.PROCEDURE-FONCTION
NOM.PARAMETRE
TYPE.PARAMETRE

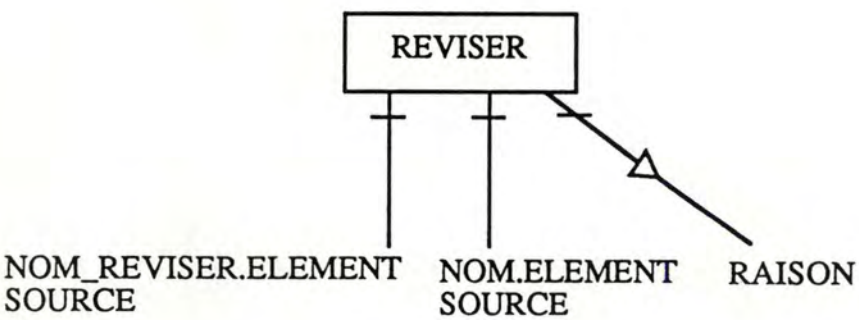

AVOIR
NOM.MODULE
NOM.PROCEDURE-FONCTION
NOM.PARAMETRE
GENRE.AVOIR

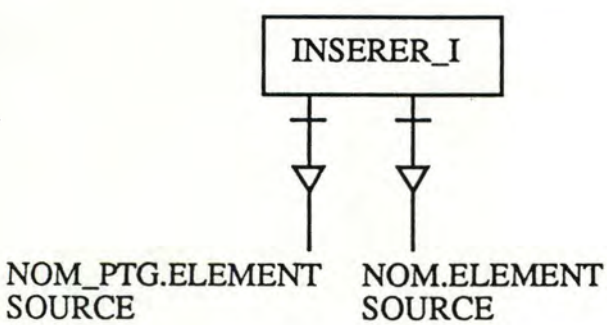

ELEMENT SOURCE
NOM.ELEMENT SOURCE
LANGAGE.ELEMENT SOURCE
GENRE.ELEMENT SOURCE

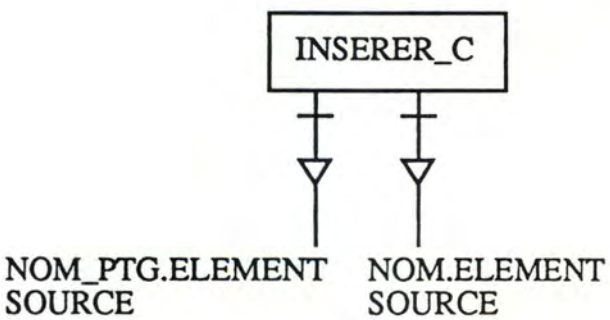

TRANSFORMER
NOM.MODULE
NOM.ELEMENT SOURCE
GENRE.ELEMENT SOURCE

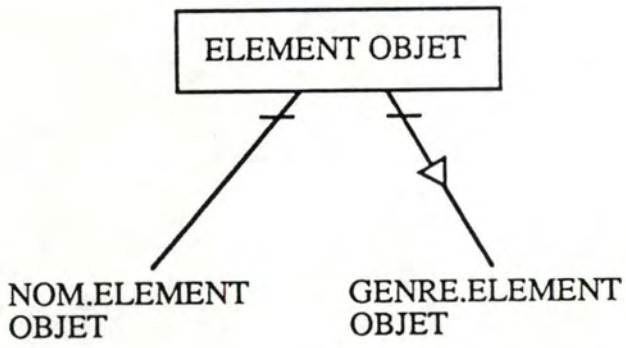

CONTENIR
NOM.LIBRAIRIE SOURCE
NOM.ELEMENT SOURCE

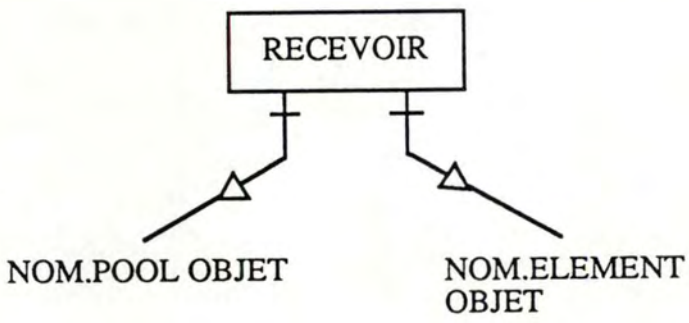

VARIER
NOM_VARIER.ELEMENT SOURCE
NOM.ELEMENT SOURCE
RAISON

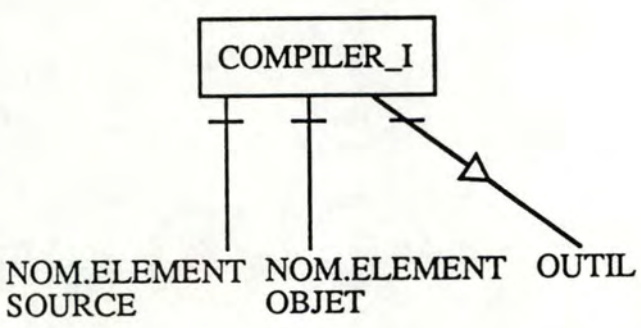

REVISER
NOM_REVISER.ELEMENT SOURCE
NOM.ELEMENT SOURCE
RAISON

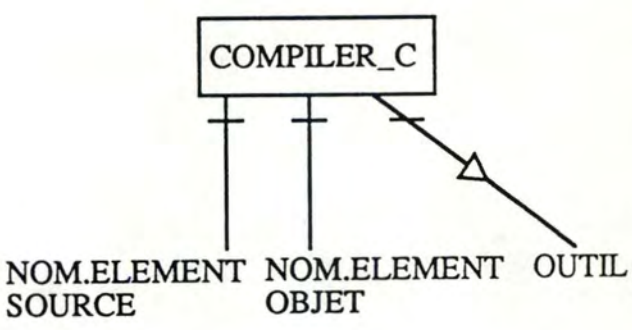

INSERER_I
NOM_PTG.ELEMENT SOURCE
NOM.ELEMENT SOURCE

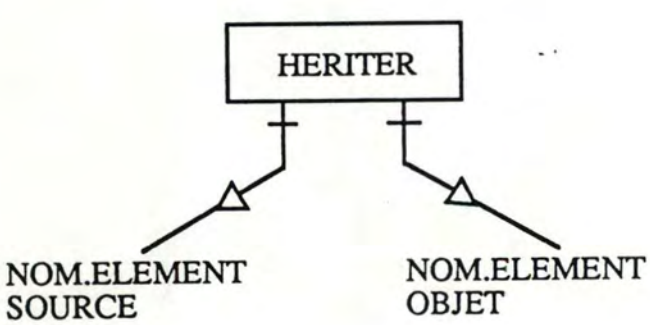

INSERER_C
NOM_PTG.ELEMENT SOURCE
NOM.ELEMENT SOURCE


ELEMENT OBJET
NOM.ELEMENT OBJET
GENRE.ELEMENT OBJET


RECEVOIR
NOM.POOL OBJET
NOM.ELEMENT OBJET


COMPILER_I
NOM.ELEMENT SOURCE
NOM.ELEMENT OBJET
OUTIL


COMPILER_C
NOM.ELEMENT SOURCE
NOM.ELEMENT OBJET
OUTIL


HERITER
NOM.ELEMENT SOURCE
NOM.ELEMENT OBJET

(Balzer 81) Robert M. Balzer.
"Transformational Implementation : An Exemple"
IEEE Transactions on Software Engineering,
Vol SE-7, n°1, January 1981. pp 3-14.

(Balzer 82) Robert M. Balzer, Neil M. Goldman et David S. Wile.
"Operational Specification as the Basis for Rapid Prototyping"
ACM Sigsoft Software Engineering Notes,
Vol 7, n°7, December 1982. pp 3-16.

(Bazelmans 85) Rudy Bazelmans.
"Evolution of Configuration Management"
ACM Sigsoft software Engineering Notes,
vol 10, n°5, Octobre 1985. pp 37-46.

(Belkhatir 88) Nouredine Belkhatir.
"Nomade : un noyau d'environnement pour la programmation globale"
Thèse de doctorat de l'Institut National Polytechnique de Grenoble, Décembre 1988.

(Bodart 83) François Bodart et Yves Pigneur.
"Conception assistée des applications informatiques"
Vol. 1 : Etude d'opportunité et analyse conceptuelle
Masson, Paris 1983.

(Boehm 76) Barry. W. Boehm.
"Software Engineering"
IEEE Transactions on Computers,
December 1976, Vol. C-25 n°12, pp 1226-1241.

(Boehm 81) Barry W. Boehm.
"Software Engineering Economics"
Prentice Hall, Inc., Englewood Cliffs,
New Jersey, 1981.

(Boehm 87) Barry W. Boehm.
"Improving Software Productivity"
Computer, Septembre 1987, pp 43-47.

(Chen 76) P. Chen.
"The Entity-Relationship Model
Toward a Unified View of Data."
ACM-TODS, Vol. 1, n°1, March 1976.

(Codd 70) E.Codd.
"A Relationnal Model of Data for Large Shared
Data Banks"
Comm. ACM, Vol. 14, n°6, pp 377-387, 1970.

(DEC/PAS 87) Digital Corporation Equipment.
"VAX PASCAL Reference Manual"
Order Number AI-L369C-TE, February 1987.

(Deville 89) Yves Deville.
"Logic Programming - Systematic Program Development"
Forthcoming book of Addison Wesley;
Reading (Massachusetts, USA), 1989.

(DOD 68)
"Configuration Control - Engineering Changes,
Deviations and Waivers"
Department of defense, 1968.

(Gallo 86) F. Gallo, R. Minot and I. Thomas.
"The Object Management System of PCTE as a Software
Engineering Database Management System"
Proceedings of the 2nd ACM Sigsoft/Sigplan Symposium
on Practical Software Development Environments,
Palo Alto (California), December 9 - 11, 1986,
ACM Sigplan Notices,
Vol. 22, n°1, pp 12 - 15, January 1987.

(Hainaut 86) Jean-Luc Hainaut.
"Conception assistée des applications informatiques"
Vol. 2 : Conception de la base de données,
Masson, Paris 1986.

(Horowitz 86) Ellis Horowitz & Ronald C. Williamson.
"SODOS : A Software Documentation Support
Environment - Its Definition"
IEEE Transactions on Software Engineering,
Vol. SE-12, n°8, pp 849 - 859, August 1986.

(IEEE 83)
"IEEE Standard Glossary of Software Engineering
Terminology"
IEEE Standard 729, 1983.

(INGRES/ESQ) Relational Technology Inc.
"INGRES/EMBEDDED SQL User's Guide and Reference Manual"
Release 5.0, VAX/VMS, August 1986.

(INGRES/PASCAL) Relational Technology Inc.
"INGRES/EMBEDDED SQL Companion Guide for Pascal"
Release 5.0, VAX/VMS, August 1986.

(INGRES/SQL) Relational Technology Inc.
"INGRES/SQL Reference Manual"
Release 5.0, VAX/VMS, August 1986.

(Leblang 88) D. Leblang, R.P. Chase.
"Parallel Building : Experience with a Case for Workstations Networks"
Int. Workshop on Software Version and Configuration Control.
27-29 January, Grassau-FRG, J.F.H. Winkler (Ed.).

(Meyer 88) Bertrand Meyer.
"Object-Oriented Software Construction"
Prentice hall, 1988.

(Meyer 85a) Bertrand Meyer.
"The Sofware Knowledge Base"
Proceedings of the 8th International Conference on Software Engineering,
London, 1985, pp 158 -165

(Osterweil 81) Leon Osterweil.
"Software Environment Research"
Directions for the Next Five Years,
IEEE Computer, Vol. 14, Number 4, pp 35 - 43,
April 1981.

(Partsch 83) H. Partsch et R. Steinbrüggen.
"Program Transformation Systems"
Computing Surveys, Vol. 15, n°3,
September 1983, pp 199-236.

(Penedo 85) Maria H. Penedo and E. Don Stuckle.
"PMDB - A project Master Data Base for Software Engineering Environments"
Proceedings of the 8th International Conference on Software Engineering,
London, 1985, pp 150 - 157.

(Penedo 87) Maria H. Penedo.
"Prototyping a Project Master Data Base for Software Engineering Environments"
Proceedings of the 2nd ACM Sigsoft/Sigplan Symposium on Practical Software Development Environments, Palo Alto (California), December 9 - 11, 1986, ACM Sigplan Notices, Volu. 22, n° 1, pp 1 - 11, January 1987.

(Rochkind 75) Marc J. Rochkind.
"The source Code Control System"
IEEE Transactions on Software Engineering, Vol. SE - 1, n°4, pp 364 -370, December 1975.

(Royce 70) W.W. Royce.
"Managing the Development of Large Software Systems : Concepts and Techniques"
Wescon Proc., August 1970.

(Scharer 83) Laura Scharer.
"The Prototyping Alternative"
ITT Programming, Voll, n°1, 1983, pp 34-43.

(Taylor 82) Tamara Taylor et Thomas A. Standish.
"Initial Thoughts on Rapid Prototyping Techniques"
ACM Sigsoft Software Engineering Notes, Vol. 7, n°5, december 1982, pp 160-166.

(Tichy 85) W.F. Tichy.
"RCS- a System for Version Control"
Software Practice and Experience, Vol. 15, n° 47, July 1985.

(Zave 82) Pamela Zave.
"An Operational Approach to Requirements Specification for Embedded Systems"
IEEE Transactions on Software Engineering, Vol. SE-8, n°3, May 1983, pp 250-269.

(van Lamsweerde 82) Axel van Lamsweerde.
"Les outils d'aide au développement de logiciels : un aperçu des tendances actuelles,
Proceedings JIIA, Paris, June 1982.

(van Lamsweerde 86) Axel van Lamsweerde et al.
"The Kernel of a Generic Software Development Environments"
Proceedings of the 2nd ACM Sigsoft/Siplan Symposium on Practical Software Development Environments,
Palo Alto (California), December 9 - 11, 1986,
ACM Sigplan Notices, Vol. 22, n°1, pp 208 - 217,
January 1987.

(van Lamsweerde 88) Axel van Lamsweerde.
"Situation de l'atelier ALMA par rapport à l'état de l'art actuel.
January 1988.

(Zave 84) Pamela Zave.
"The Operational Versus the Conventional Approach to Software Development"
Communications of the ACM,
Volume 27, n°2, February 1984, pp 104-118.

TABLE DES MATIERES.

Chapitre 1. Tendances actuelles des modèles de cycle de vie 1

1.1 Introduction 1

1.2 Des modèles de cycle de vie 4

1.2.1. Le modèle "Code and Fix."

1.2.2. Le modèle en cascade.

1.2.3. Le prototypage.

1.2.4. Les spécifications opérationnelles.

1.2.5. L'implémentation transformationnelle.

1.2.6. Conclusion.

1.3. Les ateliers logiciels 15

1.3.1. Intégration d'outils autonomes.

1.3.2. Couverture de la totalité du cycle de vie.

1.3.3. Interface agréable à l'utilisateur.

1.3.4. Atelier générique.

Chapitre 2. Modèle de cycle de vie en programmation impérative...20

2.1. Introduction 20

2.2. Modèle du cycle de vie 25

2.3. Présentation générale du schéma Entité/Relation 28

2.3.1. Le sous-schéma "architecture".

2.3.2. Le sous-schéma "source".

2.3.3. Le sous-schéma "objet".

2.4. Spécialisation fonctionnelle de l'outil 41

2.5. Solution 43

2.5.1. Mise en oeuvre de la structure.

2.5.2. Brève description de l'architecture.

2.6. Evaluation de l'outil présenté 46

Chapitre 3. Modèle de cycle de vie en programmation logique 48

3.1. Introduction 48

3.2. Modélisation du développement de programmes logique 49

3.2.1. Présentation générale de la méthodologie.

3.2.2. Raffinement de l'étape "spécification de procédure".

3.2.3. Raffinement de l'étape "description logique".

3.2.4. Raffinement de l'étape "procédure logique pure".

3.2.5. Raffinement de l'étape "code PROLOG".

3.2.6. Extension du modèle E/R.

3.3. Spécification d'un système de gestion de configurations....80

3.3.1. Le problème de l'identification des versions.

3.3.2. Mécanisme de sélection de versions.

3.3.3. Le contrôle des versions.

3.4. Evaluation du travail 95

Conclusion et perspectives 97

1. Aspects abordés.

2. Contributions personnelles.

3. Perspectives.

Annexes 99

Références bibliographiques 108

FM B16/1989/86

FACULTES
UNIVERSITAIRES
N.D. DE LA PAIX
NAMUR

INSTITUT D'INFORMATIQUE

CONTRIBUTION A L'ETUDE DE MODELES DE CYCLE DE VIE EN PROGRAMMATION IMPERATIVE ET LOGIQUE.

Philippe De Raedemacker
Promoteur : Professeur Baudouin Le Charlier

Plan de l'exposé

1. Situation du travail réalisé

2. Présentation d'un sujet original

3. Evaluation du travail

1. Situation du travail réalisé

Framework identique :

- présentation d'une méthodologie de programmation,

- modélisation,

- spécification d'outil.

2. Présentation d'un sujet original

2.1. Définitions

- Définition générale

- Définition affinée

2.2. Présentation de quelques fonctionnalités

- Identification des objets

- Sélection des objets

- Contrôle des objets

    Gestion de l'espace d'archivage

    Gestion d'historique

    Gestion des modifications concurrentes

    Fusion d'objets

2.1. Définitions

- **Définition générale**

La *gestion de configuration* :

. l'organisation,

. la documentation,

. le suivi

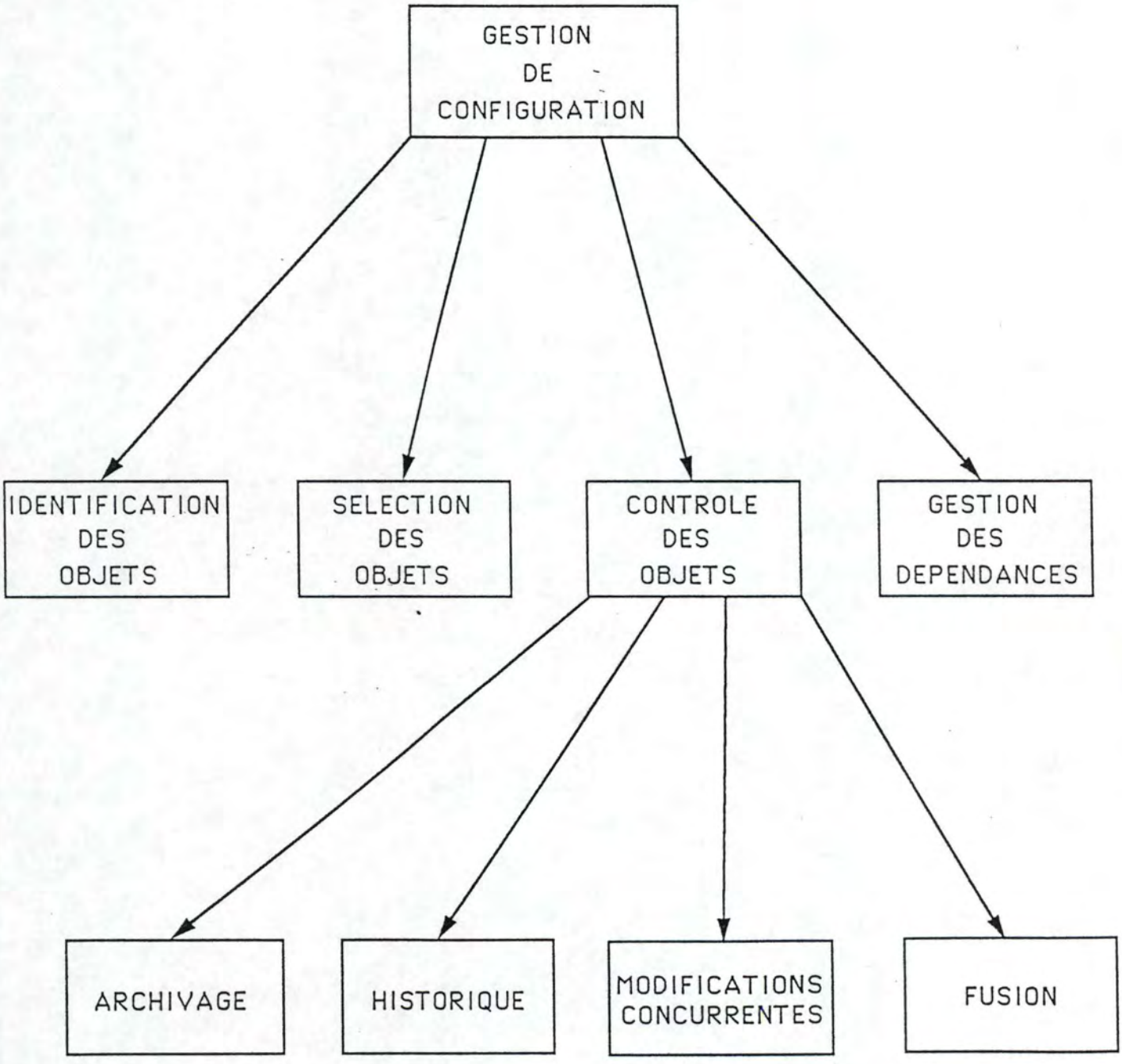
des objets d'un système logiciel.

Un *objet* d'un système logiciel :

un des composants produits durant le cycle de vie d'un sytème logiciel.

2.1. Définitions

- Définition affinée



Légende :

⟶ = relation " se compose de "

2.2. Présentation de quelques fonctionnalités

- Identification des objets

  - Notion de révision d'un objet :

    développement linéaire d'un objet

    exemples :

      raffinements successifs

      corrections d'erreurs

  - Notion d'alternative d'un objet :

    développement parallèle et indépendant d'un objet

    exemples :

      alternatives fonctionnelles

      alternatives d'implémentation

- Système d'identification :

(nom_objet, alternative, révision)

exemple :

fusion_liste /pas : turbo .15

2.2. Présentation de quelques fonctionnalités

- **Sélection des objets**

  - sélection en extension :

    énumération complète de la version.

    exemple :

    fusion_liste /pas : turbo .15

  - sélection par défaut :

    choix implicite pour des champs omis.

    exemple :

    fusion_liste /pas

  - sélection en intention :

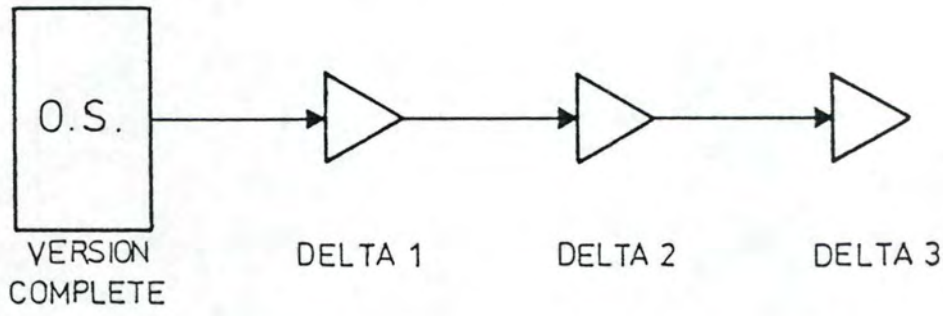
    sélection selon des valeurs
    d'attributs des objets.

2.2. Présentation de quelques fonctionnalités

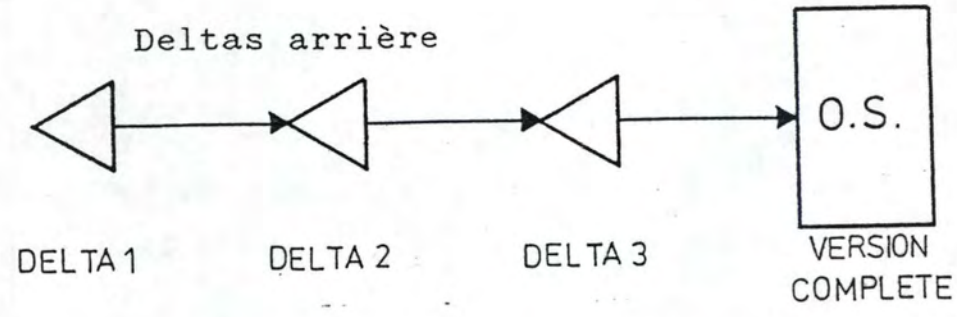
- Contrôle des objets

Gestion de l'espace d'archivage :
le mécanisme des deltas

Les deltas avant :



Les deltas arrière :

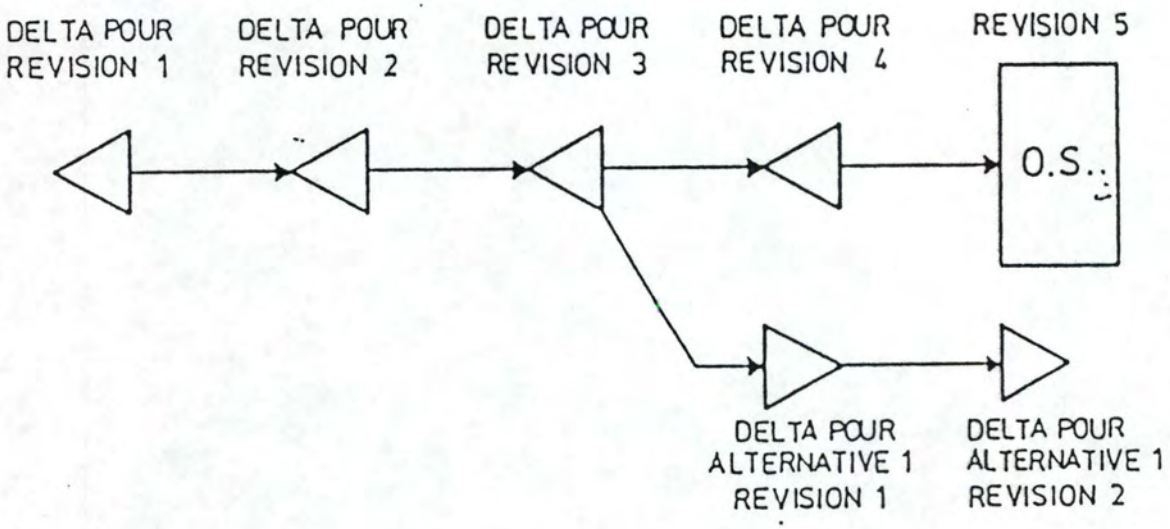


Les deltas interfoliés :

un seul fichier séquentiel constitué :

- blocs de texte,
- blocs de contrôle.

Solution choisie :



2.2. Présentation de quelques fonctionnalités

- Contrôle des objets

Gestion d'historique :

enregistrement d'informations concernant les modifications.

exemples :

- nom du responsable
- date et heure
- commentaires

2.2. Présentation de quelques fonctionnalités

- Contrôle des objets

Gestion des modifications concurrentes :

mécanismes de

- réservation d'objet,
- libération d'objet.

2.2. Présentation de quelques fonctionnalités

- **Contrôle des objets**

  Fusion d'objets :

  fusion des modifications effectuées en parallèle...

  --> pas de solution générale !

3. Evaluation du travail

- intégration des outils dans un environnement complet,

- choix d'implémentation de la BD,

- construction des outils de gestion de version,

- développement de la gestion des dépendances.

Memoire De Knedemaeker

1989.90

FM B16/1989/36

Erratum.

Veuiller lire à la :

- page 2, troisième paragraphe :

  sa compatibilité au lieu de sa comptabilité

- page 12, troisième paragraphe :

  l'expression de spécification au lieu de l'expansion de spécification

- page 16, point 1.3.1.1. :

  chaque élément d'un n-uple au lieu de chaque élément d'un cycle

- page 38, deuxième paragraphe :

  un élément objet au lieu d'un élément source

- page 41, deuxième paragraphe :

  le texte corps sera dans au lieu de le texte corps sera dans

- page 41, dernier paragraphe :

  aux textes interface au lieu de aux textes source

- page 53, dernier paragraphe ajouter :

  les paramètres

- page 57, premier paragraphe,

  est caractérisée par au lieu de est considérée par

- page 61, dernier paragraphe,

  la sémantique procédurale au lieu de la logique sémantique procédura

- page 62, premier paragraphe,

  une formule bien fondée au lieu d'une formule bien fermée

- page 65, seconde contrainte,

  des relation bien fondées au lieu de des relations bien fondées

- page 69, premier paragraphe,

  Nd dérivations sont au lieu de Nd sont

. page 73, premier paragraphe :

pour une spécification au lieu de pour une signification

. page 82, premier paragraphe :

type = dl désigne un objet description logique

au lieu de type = dl désigne un objet procédure logique

. page 86, début de page :

3.3.2.3. Sélection en intention au lieu de Définition en intention

. page 90, premier paragraphe :

un accès à la dernière révision est demandé. Nous choisirons,

dès lors, de les gérer selon la technique des deltas arrière.

. page 91, premier paragraphe :

une technique de compactage des blancs au lieu d'une technique

de comparaison des blancs

. page 95,

3.4. Evaluation du travail au lieu de 3.2. Evaluation du travail

. page 95, dernier paragraphe :

les objets atteints par au lieu de les objectifs atteints par